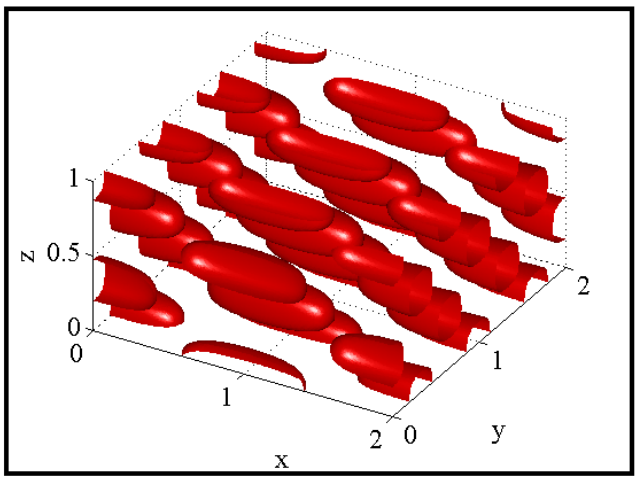
University College Dublin
An Coláiste Ollscoile, Baile Átha Cliath

School of Mathematical Sciences
Scoil na nEolaíochtaí Matamaitice
Summer School in mini-HPC



Dr Lennon Ó Náraigh

Lecture notes in mini-HPC, May 2013

Summer School in mini-HPC

Scope of summer school

With increasing computational power available even on desktop level, opportunities for simulating large-scale physical, biological and social systems are ubiquitous. Moreover, as the scale and scope of simulations increase, the challenge of handling ever-larger data sets becomes immense. This challenge is amplified by the phenomenon of "big data" – the generation of more and more data as the internet permeates our lives, and the possibility of a data wall, should the rate of data generation outstrip our capacity to interpret this information. The object of this module is to study high-performance computing at elementary level (along with standard data I/O) operations, and to interpret the resulting large data files in an automated fashion using batch-processing techniques that can be implemented in Matlab.

Learning Outcomes

1. Be able to write elementary programs in Fortran, including subroutines, and compiler linking
2. Be able to handle data I/O in Fortran
3. Be able to implement multi-threading in Fortran
4. Be familiar with the theory of multi-threading, the pitfalls in its implementation, and its limitations with respect to computer architecture
5. Be able to automate data post-processing from Fortran simulations in Matlab, including Matlab I/O, visualization (e.g. 3D isosurfaces, contour plotting, animations), and statistical post-processing.
6. Be familiar with the pitfalls associated with data-processing of weakly-structured files

Prerequisites

Students must have taken ACM 20030 or its non-UCD equivalent and have a working knowledge of Matlab. No prior knowledge of Fortran of the Linux operating system is assumed, although some knowledge of these areas would be helpful.

Assessment No assessment takes place in the summer school. Students who complete the school satisfactorily will receive a certificate of attendance.

Editions

First edition: May 2013

Contents

Module description i

1 Introduction 1

2 Floating-Point Arithmetic 4

3 Computer architecture and Compilers 16

4 The model diffusion equation – theoretical background 23

5 The model Poisson problem 31

6 Model diffusion equation – Numerical setup 34

7 The model Poisson problem – numerical setup 39

8 Jacobi iteration – convergence 43

9 Successive over-relaxation 51

10 Introduction to Fortran 56

11 Challenge problem in Fortran 68

12 Introduction to shared memory 76

13 Multithreading for the model Poisson equation 83

14 Memory allocation in Fortran 94

15 Handling data in large files **97**

A Facts about Linear Algebra you should know **105**

Chapter 1

Introduction

Module summary

We will follow the following programme of work:

1. Study a little bit of computer architecture. This is because we want to understand the concepts behind parallel programming, and then implement them.

2. Study some analytical models. These will be the test cases on which we do numerical work. The analytical models include a diffusion equation and Poisson's equation, in two dimensions, with a mixture of boundary conditions.

3. Study some numerical methods, in particular central differencing and the Crank–Nicholson scheme for diffusion.

4. Study a Matlab code to solve Poisson's equation in two dimensions, on a rectangular domain. You will extend this code to a diffusion problem.

5. Study rigorously iterative methods for solving matrix problems, investigating readily-computable sufficient conditions for the convergence of such methods.

6. Learn the basics of Fortran, including code-writing, compilation, execution, and I/O (input/output).

7. Study a **Fortran** code to solve Poisson's equation in two dimensions, on a rectangular domain. You will extend this code to a diffusion problem.

8. Write postprocessing routines in Matlab to examine and visualize outputs from Fortran codes.

9. Study the concept of shared memory and its implementation via the OpenMP (OMP) standard.

10. Parallelize our Fortran codes.
11. Investigate interesting conceptual issues in parallel programming: race conditions, reductions, thread-local operations, memory use.
12. Assess the performance of a parallel code by timing the execution of such codes.
13. Study issues around memory allocation in Fortran.
14. Write, run, and postprocess a three-dimensional code for solving Poisson's problem in a regular three-dimensional domain.

Textbooks

- Lecture notes are available for download on the web and are self-contained:

  `http://mathsci.ucd.ie/~onaraigh/summerschool.html`

- The lecture notes will also be used as a practical Matlab/Fortran guide in the lab-based sessions.
- Here is a list of the resources on which the notes are based:
  - For issues concerning numerical linear algebra:

    `http://tavernini.com/arc/mat5603note2.pdf`

    as well as

    `http://www.math.drexel.edu/~foucart/TeachingFiles/F12/M504Lect6.pdf`
  - For issues concerning computer architecture and memory, the course *Introduction to High-Performance Scientific Computing* on the website

    `www.tacc.utexas.edu/~eijkhout/Articles/EijkhoutIntroToHPC.pdf`
  - For more information about the numerical solution of partial differential equations, the book *Numerical Recipes in C*, W. H. Press *et al.* (CUP, 1992):

    `http://apps.nrbook.com/c/index.html`
  - For parallel programming in the OpenMP (OMP) standard, the lecture notes *Parallel Programming in Fortran 95 using OpenMP*, M. Hermanns (2002):

    `http://www.openmp.org/presentations/miguel/F95_OpenMPv1_v2.pdf`

Module dependencies

Some knowledge of Linear Algebra and Calculus is assumed. Students must have taken ACM 20030 or its non-UCD equivalent and have a working knowledge of Matlab. No prior knowledge of Fortran of the Linux operating system is assumed.

School format

- Mornings, 10:00 (sharp) – 12:30 (ish): Lectures
- Afternoons, 14:00 (sharp) – 16:00: supervised lab and problem sessions

Of course, you can stay after 16:00 (until the building closes) to finish problems.

Chapter 2

Floating-Point Arithmetic

Overview

Binary and decimal arithmetic, floating-point representation, truncation, truncation errors, IEEE double precision standard

2.1 Introduction

Being electrical devices, 'on' and 'off' are things that all computers understand. Imagine a computer made up of lots of tiny switches that can either be on or off. We can represent any number (and hence, any information) in terms of a sequence of switches, each of which is in an 'on' or 'off' state. We do this through **binary arithmetic**. An 'on' or an 'off' switch is therefore a fundamental unit of information in a computer. This unit is called a **bit**.

2.2 Positional notation and base 2

One of the crowing achievements of human civilization is the ability to represent arbitrarily large and small **real numbers** in a compact way using only ten digits. For example, the integer $570,123$ really means

$$570,123 = (5 \times 10^5) + (7 \times 10^4) + (0 \times 10^3) + (1 \times 10^2) + (2 \times 10^1) + (3 \times 10^0)$$

Here,

- The leftmost digit (5) has five digits to its right and therefore comes with a power 10^5,

- The digit second from the left (7) has four digits to its right and therefore comes with power of 10^4,

- And so on, down to the rightmost digit, which, by definition, has no other digits to its right, and therefore comes with a power of 10^0.

In contrast, the Romans would have struggled to represent this number:

$$570,123 = \overline{\mathrm{D}}\,\overline{\mathrm{L}}\,\overline{\mathrm{X}}\,\overline{\mathrm{X}}\,\mathrm{C}\,\mathrm{X}\,\mathrm{X}\,\mathrm{I}\,\mathrm{I}\,\mathrm{I},$$

where the overline means multiplication by $1,000$.

Rational numbers with absolute value less than unity can be expressed in the same way, e.g. 0.217863:

$$0.217863 = (2\times10^{-1}) + (1\times10^{-2}) + (7\times10^{-3}) + (8\times10^{-4}) + (6\times10^{-5}) + (3\times10^{-6}).$$

Other rational numbers have a decimal expansion that is infinite but consists of a periodic repeating pattern of digits:

$$\begin{aligned}\tfrac{1}{7} = 0.142857142857\cdots &= (1\times10^{-1})+(4\times10^{-2})+(2\times10^{-3})+(8\times10^{-4})+(5\times10^{-5})+(7\times10^{-6})\\ &+ (1\times10^{-7}) + (4\times10^{-8}) + (2\times10^{-9}) + (8\times10^{-10}) + (5\times10^{-11}) + (7\times10^{-12}) + \cdots\end{aligned}$$

Using geometric progressions, it can be checked that $1/7$ does indeed equal $0.142857142857\cdots$, since

$$\begin{aligned}0.142857142857\cdots &= 1\left(\frac{1}{10}+\frac{1}{10^7}+\frac{1}{10^{13}}+\cdots\right)+4\left(\frac{1}{10^2}+\frac{1}{10^8}+\cdots\right)+\\ &\quad +2\left(\frac{1}{10^3}+\frac{1}{10^9}+\cdots\right)+8\left(\frac{1}{10^4}+\frac{1}{10^{10}}+\cdots\right)+\\ &\quad +5\left(\frac{1}{10^5}+\frac{1}{10^{11}}+\cdots\right)+7\left(\frac{1}{10^6}+\frac{1}{10^{12}}+\cdots\right)+\cdots\\ &= \frac{1}{10}\left(1+\frac{1}{10^6}+\frac{1}{10^{12}}+\cdots\right)+\frac{4}{10^2}\left(1+\frac{1}{10^6}+\frac{1}{10^{12}}\right)+\cdots\\ &= \left(1+\frac{1}{10^6}+\frac{1}{10^{12}}+\cdots\right)\left[\frac{1}{10}+\frac{4}{10^2}+\frac{2}{10^3}+\frac{8}{10^4}+\frac{5}{10^5}+\frac{7}{10^6}\right]\\ &= \frac{1}{1-\frac{1}{10^6}}\left(\frac{10^5+5\times10^4+2\times10^3+8\times10^2+5\times10+7}{10^6}\right),\end{aligned}$$

Hence,

$$\begin{aligned} 0.142857142857\cdots &= \frac{10^6}{10^6-1}\left(\frac{10^5+5\times10^4+2\times10^3+8\times10^2+5\times10+7}{10^6}\right), \\ &= \frac{10^5+4\times10^4+2\times10^3+8\times10^2+5\times10+7}{10^6-1}, \\ &= \frac{142857}{999999}, \\ &= \frac{142857}{7\times142857}, \\ &= \tfrac{1}{7}. \end{aligned}$$

In a similar way, all real numbers can be represented as a decimal string. The decimal string may terminate or be periodic (rational numbers), or may be infinite with no repeating pattern (irrational numbers). For example, a real number $y \in [0,1)$, with

$$y = \sum_{n=1}^{\infty} \frac{x_n}{10^n} = 0.x_1x_2\cdots$$

where $x_i \in \{0,1,\cdots,9\}$. This number does not as yet have a meaning. However, consider the sequence $\{y_N\}$ of rational numbers, where

$$y_N = \sum_{n=1}^{N} \frac{x_n}{10^n}. \tag{2.1}$$

This is a sequence that is bounded above and monotone increasing. By the **completeness axiom**, the sequence has a limit, hence

$$y = \lim_{N\to\infty} y_N.$$

The completeness axiom is therefore equivalent to the construction of the real numbers: any real number can be obtained as the limit of a rational sequence such as Equation (2.1).

Now that we understand how numbers are represented in base 10 using positional notation, we now examine other bases. Consider for example the string

$$x = 1010110,$$

in base 2. Using positional notation and base 2, we understand x to be the number

$$\begin{aligned} x &= (1\times2^6)+(0\times2^5)+(1\times2^4)+(0\times2^3)+(1\times2^2)+(1\times2)+(0\times2^0), \\ &= 64+16+4+2, \\ &= 86, \text{ base } 10. \end{aligned}$$

Numbers with absolute value less than unity can be represented in a similar way. For example, let

$$x = 0.01101 \text{ base } 2.$$

Using positional notation, this is understood as

$$\begin{aligned} x &= \frac{0}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{0}{2^4} + \frac{1}{2^5}, \\ &= \tfrac{1}{4} + \tfrac{1}{8} + \tfrac{1}{32}, \\ &= \tfrac{8}{32} + \tfrac{4}{32} + \tfrac{1}{32}, \\ &= \tfrac{13}{32}, \\ &= 0.40625 \text{ base } 10. \end{aligned}$$

Two binary strings can be added by 'carrying twos'. For example,

$$\begin{array}{rr} & 0.01101 \\ + & 1.11100 \\ \hline & 10.01001 \end{array}$$

Let's check our calculation using base 10:

$$\begin{aligned} x_1 &= 0.01101 = \frac{0}{2} + \frac{1}{4} + \frac{1}{8} + \frac{0}{16} + \frac{1}{32} = \frac{13}{32}, \\ x_2 &= 1.111 = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} = \frac{15}{8} = \frac{60}{32}. \end{aligned}$$

Hence,

$$x_1 + x_2 = \frac{73}{32} = 2 + \frac{9}{32} = 2 + \frac{1}{32} + \frac{8}{32} = 2 + \frac{1}{32} + \frac{1}{4} = (1 \times 2^1) + (0 \times 2) + \frac{1}{2^2} + \frac{1}{2^5} = 10.01001 \text{ base } 2.$$

Because computers (at least notionally) consist of lots of switches that can be on or off, it makes sense to store numbers in binary, as a collection of switches in 'on' or 'off' states can be put into a one-to-one correspondence with a set of binary numbers. Of course, a computer will always contain only a **finite** number of switches, and can therefore only store the following kinds of numbers:

1. Numbers with absolute value less than unity that can be represented as a binary expansion with a finite number of non-zero digits;

2. Integers less than some certain maximum value;

3. Combinations of the above.

An irrational real number (e.g. $\sqrt{2}$) will be represented on a computer by a truncation of the true value. This introduces a potential source of error into numerical calculations – so-called **rounding error**.

2.3 Floating-point representation

Rounding error is the original sin of computational mathematics. A partial atonement for this sin is the idea of **floating-point arithmetic**. A base-10 floating-point number x consists of a fraction F containing the significant figures of the number, and an exponent E:

$$x = F \times 10^E,$$

where

$$\tfrac{1}{10} \leq F < 1.$$

Representing floating-point numbers on a computer comes with two kinds of limitations:

1. The range of the exponent is limited, $E_{\min} \leq E \leq E_{\max}$, where $E_{\min}$ is negative and $E_{\max}$ is positive; both have large absolute values. Calculations leading to exponents $E > E_{\max}$ are said to lead to **overflow**; calculations leading to exponents $E < E_{\min}$ are said to have **underflowed**.

2. The number of digits of the fraction F that can be represented by on and off switches on a computer is finite. This results in rounding error.

The idea of working with rounded floating-point numbers is that the number of significant figures ('precision') with which an arbitrary real number is represented is independent of the magnitude of the number. For example,

$$x_1 = 0.0000001234 = 0.1234 \times 10^{-6}, \qquad x_2 = 0.5323 \times 10^6$$

are both represented to a precision of four significant figures. However, let us add these numbers, keeping only four significant figures:

$$\begin{aligned} x_1 + x_2 &= 0.0000001234 + 532,300, \\ &= 532,300.0000001234, \\ &= 0.5323000000001234 \times 10^6, \\ &= 0.5323 \times 10^6 \qquad \text{four sig. figs.}, \\ &= x_1. \end{aligned}$$

Rounding has completely negated the effect of adding x_1 and x_2.

When starting with a real number x with a possibly indefinite decimal expansion, and representing it floating-point form with a finite number of digits in the fraction F, the rounding can be implemented in two ways:

1. Rounding up, e.g.

$$0.12345 = 0.1235, \qquad \text{four sig. figs.},$$

and $0.12344 = 0.1234$ and $0.12346 = 0.1235$, again to four significant figures;

2. 'Chopping', e.g.

$$0.12345 = 0.12344 = 0.12346 = 0.1234, \qquad \text{truncated to four sig. figs.}$$

The choice between these two procedures appears arbitrary. However, consider

$$x = a.aaaaB,$$

which is rounded up to

$$\tilde{x} = a.aaaC,$$

If $B < 5$, then $C = a$, hence

$$x - \tilde{x} = 0.0000B = B \times 10^{-5} < 5 \times 10^{-5}.$$

On the other hand, if $B \geq 5$, then $C = a + 1$ (the digit is incremented by one). In a worst-case scenario, $B = 5$, and

$$\tilde{x} - x = a.aaaC - a.aaaaaB = (C - a) \times 10^{-4} - B \times 10^{-5} = 10^{-4} - 5 \times 10^{-5} = 5 \times 10^{-5}.$$

In either case therefore,

$$|\tilde{x} - x| \leq 5 \times 10^{-5}.$$

Assuming $a \neq 0$, we have $|x| > 1$, hence $1/|x| < 1$, and

$$\frac{|\tilde{x} - x|}{|x|} \leq 5 \times 10^{-5} = \tfrac{1}{2} \times 10^{-4}.$$

More generally, rounding x to N decimal digits gives a relative error

$$\frac{|\tilde{x} - x|}{|x|} \leq \tfrac{1}{2} \times 10^{-N+1}.$$

See if you can show by similar arguments that for chopping, the relative error is twice as large than that for rounding:

$$\frac{|\tilde{\tilde{x}} - x|}{|x|} \leq 10^{-N+1}.$$

A more convenient way of summarizing these results is as follows: Let

$$\tilde{x} = \text{fl}(x)$$

be the result of rounding the real number x using either rounding up or chopping. Define the signed relative error

$$\epsilon = \frac{\text{fl}(x) - x}{x}. \tag{2.2}$$

We know,

$$|\epsilon| \leq \epsilon_N = \begin{cases} \frac{1}{2}10^{-N+1}, & \text{rounding up,} \\ 10^{-N+1}, & \text{chopping.} \end{cases} \tag{2.3}$$

Thus, by definition,

$$|\epsilon| \leq \epsilon_N$$

Re-arranging Equation (2.2), we have

$$\text{fl}(x) = x(1+\epsilon), \qquad |\epsilon| \leq \epsilon_N.$$

The value ϵ_N is called **machine epsilon** and depends on the floating-point arithmetic of the machine in question. We can also think of machine epsilon as the largest number x for which the computed value of $1 + x$ is 1. It can be computed as follows in Matlab:

```
x=1;
while( 1+x~=1)
  x=x/2;
end
x=2*x;
display(x)
```

However, Matlab will display machine epsilon if you simply enter 'eps' at the command prompt.

2.4 Error accumulation

Most computing standards will have the following property:

$$\mathrm{fl}(a \circ b) = (a \circ b)(1 + \epsilon), \qquad |\epsilon| \leq \epsilon_N, \tag{2.4}$$

where ϵ_N is the machine epsilon and $\circ$ represents an arithmetic operation such as $\times$, $+$, $-$, or $\div$. This is a good property to have: if the error in representing the numbers a and b is small, then the error in representing their sum is also small. Because machine epsilon is very small, the compound error obtained in a long sequence of arithmetic operations (where each component operation has the property (2.4)) is very small. Errors induced by compounding individual errors such as Equation (2.4) are therefore almost always negligible. However, error accumulation can still occur in two other ways:

1. The numbers entered into the computer code lack the precision required for a long calculation, and 'cancellation errors' occur;

2. Certain iterative algorithms contain stable and unstable solutions. The unstable solution is not accessed if the 'initial condition' is zero. However, if the initial condition is ϵ_N, then the unstable solution can grow over time until it swamps the other, desired solution.

These sources of error will become more apparent in the examples in the homework.

2.5 Double precision and other formats

The gold standard for approximating an arbitrary real number in rounded floating-point form

$$x = F \times 2^E \tag{2.5}$$

is the so-called **IEEE double precision**. A double-precision number on a computer can be thought of as a 64 contiguous pieces of memory (64 bits). One bit is reserved for the sign of the number, eleven bits are reserved for the exponent (naturally stored in base 2), and the remaining fifty-two bits are reserved for the significand. Thus, in IEEE double precision, a real number is approximated and then stored as follows:

$$x \approx \mathrm{fl}(x) = (-1)^{\mathrm{sign}} \left(1 + \sum_{i=1}^{52} \frac{b_{-i}}{2^i}\right) \times 2^{E_\mathrm{s} - 1023}.$$

Here, the exponent E_s is stored using a contiguous eleven-bit binary string, meaing that E_s can in principle range from $E_\mathrm{s} = 0$ to $E_\mathrm{s} = 2047$. However, $E_\mathrm{s} = 0$ is reserved for underflow to zero, and

$E_{\mathrm{s}} = 2047$ is reserved for overflow to infinity, meaning that the maximum possible finite exponent is $E_{\mathrm{s}} = 2046$. Accounting for offset, the maximum true exponent is

$$E = E_{\mathrm{s,max}} - 1023 = 2046 - 1023 = 1023.$$

Hence, $x_{\max} \approx 2^{1023}$. Similarly,

$$x_{\min} = 2^{1-1023} = 2^{-1022}.$$

Now, recall the formula

$$\frac{|x - \mathrm{fl}(x)|}{|x|} := \epsilon \leq \epsilon_N = \begin{cases} \frac{1}{2}10^{-N+1}, & \text{rounding up,} \\ 10^{-N+1}, & \text{chopping,} \end{cases}$$

which gave the truncation error in base 10 for truncation after N figures of significance. Going over to base two and chopping, we have

$$\frac{|x - \mathrm{fl}(x)|}{|x|} := \epsilon \leq \epsilon_N = 2^{-N+1}.$$

In IEEE double precision, the precision is $N = 52 + 1$ (the extra 1 comes from the digit stored implicitly), hence

$$\epsilon_N = 2^{-53+1} = 2^{-52}.$$

Equivalently, the smallest positive number strictly greather than 1 detectable in this standard is

$$1 + \frac{0}{2} + \frac{0}{2^2} + \cdots + \frac{1}{2^{52}},$$

and again, $\epsilon_N = 2^{-52}$ gives machine precision. The IEEE standard also supports extensions to the real numbers, including the symbols Inf (which will appear when a code has overflowed), and NaN. The symbol NaN will appear as a code's output if you do something stupid. Examples in Matlab sytanx include the following particularly egregious one:

```
x=0/0;
display(x)
```

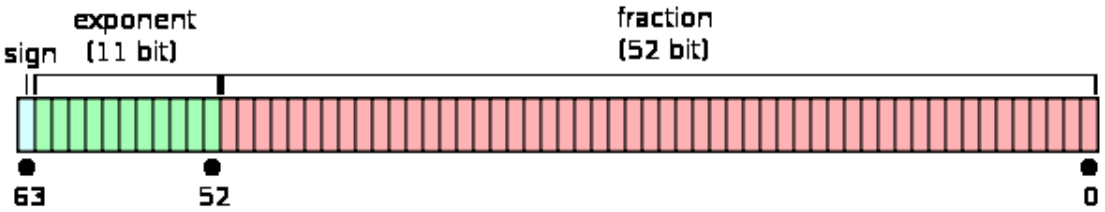


Figure 2.1: 64 contiguous bits in memory make up an IEEE floating-point number, with bits reserved for the sign, the exponent, and the fraction. From http://en.wikipedia.org/wiki/Double-precision_floating-point_format (20/11/2012).

Exercise 2.1 *If Planck's constant is* $\sim 10^{-34}$ *(SI units), and if machine epsilon is* $\sim 10^{-16}$*, how can Planck's constant be represented in IEEE double precision?*

Solution: The smallest number in absolute value terms representable in IEEE double precision is approximately 2^{-1022}, which comfortably takes in Planck's constant. This is the concept of **range**: numbers within the range $2^{-1022} \lesssim x \lesssim 2^{1023}$ can be represented.

On the other hand, machine epsilon is concerned with the concept of **precision**. Machine epsilon can be thought of as the smallest positive number ϵ_N for which the difference between $1 + x$ and 1 is detectable by the computer. Equivalently, machine precision can be thought of as an upper bound on the relative error between a real number x and its IEEE floating-point representation:

$$\frac{|x - \mathrm{fl}(x)|}{|x|} \leq \epsilon_N.$$

These concepts come together when you try to add $\hbar$ to 1 on a computer – although hbar is definitely not zero (nor will it appear as zero to the computer), the computer addition of $1 + \hbar$ will always yield 1. We are limited in the precision with which we can represent the sum $1 + \hbar$.

Another datatype is the **integer**, which is stored in a contiguous chunk of memory like a double, typically of length 8, 16, 32, or 64 bits. Typically, the integers are defined with respect to an offset (**two's complement**), so that no explicit storage of the sign is required.

Common Programming Error:

Mixing up integers and doubles. For example, suppose in a computer-programming language such as Fortran, that x has been declared to be a double-precision number. Then, assigning x the value 1, i.e.

```
x=1;
```

confuses the compiler, as it now thinks that x is an integer! In order not to confuse the compiler, one would have to write

```
x=1.d0; (Fortran)
```

Now, the distinction between integers and doubles is not enforced in Matlab, and ambiguity about variable types is allowed. However, in the Fortran part of this course, the compiler will punish you severely for this ambiguity.

Matlab implements the IEEE standard, albeit implicitly. For example, if you type

```
display(pi)
```

at the command line, you will only see the answer

```
3.1416
```

However, you can rest assured that the built-in working precision of the machine is 53 bits. For example, typing

```
display(eps)
```

yields

```
2.2204e-016
```

Also, typing

```
x=2;
while(x~=Inf)
  x_old=x;
  x=2*x;
end
display(x_old)
```

yields

```
8.9885e+307,
```

the same as $2^{1023} = 8.9885e + 307$.

Chapter 3

Computer architecture and Compilers

Overview

Computer architecture means the relationship between the different components of hardware in a computer. In this chapter, this idea is discussed under the following headings: the memory/processor model, memory organization, processor organization, simple assembly language.

3.1 The memory/processor or von Neumann model

Computer architecture means the relationship between the different components of hardware in a computer. On a very high level of abstraction, many architectures can be described as **von Neumann architectures**. This is a basic design for a computer with two components:

1. An undivided memory that stores both program and data;
2. A processing unit that executes the instructions of the program and operates on the data (CPU).

This design is different from the earliest computers in which the program was hard-wired. It is also very clever, as the line between 'data' and 'program' can become blurred – to our advantage. When we write a program in a given language, we work with a computer that has other, more basic programs installed – including a **text editor** and a **compiler**. The von Neumann architecture enables the computer to treat the code we write in the text editor as data, and the compiler is in this context a 'super-program' that operates on these data and converts our **high-level code** into instructions that can be read by the machine. Having said this, in this course, we understand 'data' to be the collection of numbers to be operated on, and the code is the set of instructions detailing the operations to be performed.

In conventional computers, the machine instructions generated by the compiled version of our code do not communicate directly with the memory. Instead, information about the location of data in the computer memory, and information about where in memory the results of data processing should go, are stored directly in a part of the CPU called the **register**. Rather counter-intuitively, the existence of this 'middle-man' register speeds up execution times for the code. Many computer programs possess **locality of reference**: the same data are often accessed repeatedly. Rather than moving these frequently-used data to and from memory, it is best to store them locally on the CPU, where they can be manipulated at will.

The main statistic that is quoted about CPUs is their Gigahertz rating, implying that the speed of the processor is the main determining factor of a computer's performance. While speed certainly influences performance, memory-related factors are important too. To understand these factors, we need to describe how computer memory is organized.

3.2 Memory organization

Practically, a pure von Neumann architecture is unrealistic because of the so-called *memory wall*. In a modern computer, the CPU performs operations on data on timescales much shorter than the time required to move data from memory to the CPU. To understand why this is the case, we need to study how the CPU and the computer memory communicate.

In essence, the CPU and the computer memory communicate via a load of wires called the **bus**. The **front-side bus** (FSB) or 'North bridge' connects the computer main memory (or 'RAM') directly to the CPU. The bus is typically much slower than the processor, and operates with clock frequencies of $\sim 1\mathrm{GHz}$, a fraction of the CPU clock frequency. A processor can therefore consume many items of data fed from the bus in one clock tick – this is the reason for the memory wall.

The memory wall can be broken up further in two parts. Associated with the movement of data are two limitations: the **bandwidth** and the **latency**. During the execution of a process, the CPU will request data from memory. Stripping out the time required for the actual data to be transferred, the time required to process this request is called latency. Bandwidth refers to the amount of data that can be transferred per unit time. Bandwidth is measured in $\mathrm{bytes/second}$, where a byte (to be discussed below) is a unit of data. In this way, the total time required to for the CPU to request and receive n bytes from memory is

$$T(n) = \alpha + \beta n,$$

where α is the latency and β is the inverse of the bandwidth ($\mathrm{second/byte}$). Thus, even with infinite bandwidth ($\beta = 0$), the time required for this process to be fulfilled is non-zero.

Typically, if the chunk of memory of interest physically lies far away from the CPU, then the latency

is high and the bandwidth is low. It is for this reason that a computer architecture tries to maximize the amount of memory near the CPU as possible. For that reason, a second chunk of memory close the CPU is introduced, called the **cache**. This is shown schematically in Figure 3.1. Data needed in

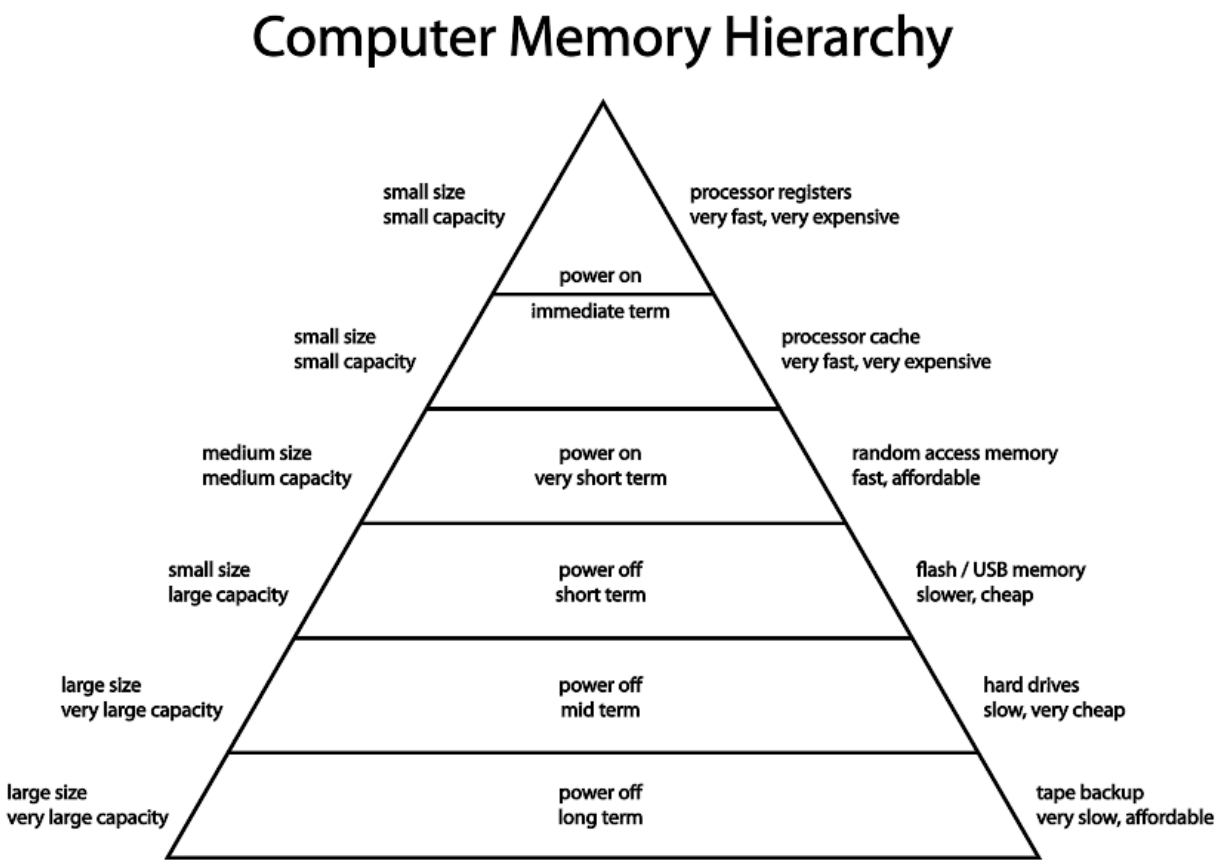


Figure 3.1: The different levels of memory shown in a hierarchy

some operation gets copied into the cache on its way to the processor. If, some instructions later, a data item is needed again, it is searched for in the cache. If it is not found there, it is loaded from the main memory. Finding data in cache is called a **cache hit**, and not finding it is called a **cache miss**. Again, the cache is a part of the computer's memory that is located *on the die*, that is, on the processor chip. Because this part of the memory is close the CPU, it is relatively quick to transfer data to and from the CPU and the cache. For the same reason, the cache is limited in size. Typically, during the execution of a programme, data will be brought from slower parts of the computer's memory to the cache, where it is moved on and off the register, where in turn, operations are performed on the data. There is a sharp distinction between the register and the cache. The instructions in machine language that have been generated by our compiled code are instructions to the CPU and hence, to the register. It is therefore possible in some circumstances to control movement of data on and off the register. On the other hand, the move from the main memory to the cache is done purely by the hardware, and is outside of direct programmer control.

3.3 The rest of the memory

The rest of the memory is referred to as 'RAM', and is neither built into the CPU (like the registers), nor collocated with the CPU (like the cache). It is therefore relatively slow but has the redeeming feature that it is large. The most-commonly known feature of RAM is that the data it contains are removed when the computer powers off. This is why you must save your work to the hard drive!

RAM itself is broken up into two parts – the **stack** and the **heap**.

Stacks are regions of memory where data is added or removed on a last-in-first-out basis. The stack really does resemble a stack of plates. You can only take a plate on or off the top of a stack – this is also true of data stored in the stack. Another silly analogy is to imagine a series of postboxes attached one on top of the other to a vertical pole. Initially, all the postboxes are empty. Then, the bottommost postbox is filled and a postit note is placed on it, indicating that the location of the next available postbox. As letters are put into and removed from postboxes, the postit note moves up and down the stack of postboxes accordingly. It is therefore very simple to know how many postboxes are full and how many are empty – a single label suffices. The system for addressing memory slots in the stack is equally simple and for that reason, accessing the stack is faster than accessing other kinds of memory.

On the other hand, there is the **heap**, which is a region of memory where data can be added or removed at will. The system for addressing memory slots in the heap is therefore much more detailed, and accessing the heap is therefore much slower than accessing the stack. However, the size of the stack is fixed at runtime and is usually quite small. Many codes require lots of memory. Trying to fit lots of data into the relatively small amount of stack that exists can lead to **stack overflow** and **segmentation faults**. Stack overflow is a specific error where the exectuting program requests more stack resources than those that exist; segmentation faults are generic errors that occur when a code tries to access addresses in memory that either do not exist, or are not available to the code. So ubiquitous and terrifying are these errors to computer codes a popular web forum for coders and computer scientists is called http://stackoverflow.com/.

For the Fortran part of this course, remember the following lesson:

Common Programming Error:

Never allocate arrays on the stack (Possibly Fatal)!

All of the different levels of memory and their dependencies are summarized in the diagram at the end of this chapter (Figure 3.2).

3.4 Multicore architectures

If you open the task manager on a modern machine running Windows, the chances are you will see two panels by first going to 'performance' and then 'CPU Usage History' . It would appear that the machine has two CPUs. In fact, modern computers contain **multiple cores**. We still consider the machine to have a single CPU, but two smaller processing units (or cores) are placed on the same chip. The two cores share some cache ('L2 cache'), while some other cache is private to each core ('L1 cache'). This enables computer to break up a computational task into two parts, work on each task separately, via the private cache, and communicate necessary shared data via the shared cache. This architecture therefore facilitates **parallel computing**, thereby speeding up computation times. High-level programs such as MATLAB take advantage of multiple-core computing without any direction from the user. On the other hand, lower-level programming standards (e.g. C, Fortran) require explicit direction from the user in order to implement multiple-core processing. This is done using the OpenMP standard.

Unfortunately, the idea of having several cores on a single chip makes the description of this architecture ambiguous. We reserve the word **processor** for the entire chip, which will consist of multiple sub-units called **cores**. Sometimes the cores are referred to as **threads** and this kind of computing is called **multi-threaded**.

3.5 Compilers

As mentioned in Section 3.1, a standard procedure for writing code is the following:

1. Write the code in a **high-level** computer language such as C or Fortran. You will do this in a text editor. Computer code on this level has a definite syntax that is very similar to ordinary English.

2. Convert this high-level code to **machine-readable** code using a **compiler**. You can think of this as a translator that takes the high-level code (readable to us, and similar in its syntax to English) into lots of gobbledegook that only the computer can understand.

3. Compilation takes in a text file and outputs a machine-readable **executable** file. The executable can then be run from the **command line**.

MATLAB sits one level higher than a high-level computer language, with a friendly syntax and all sorts of clever procedures for allocating memory so that we don't need to worry about technical issues. It also has a user-friendly interface so that our high-level Matlab files can be run and the output interpreted and plotted in a user-friendly fashion. Incidently, Matlab is written in C, so it as

though two translations happen before the computer executes our code: Matlab→ C → (Machine-readable code).

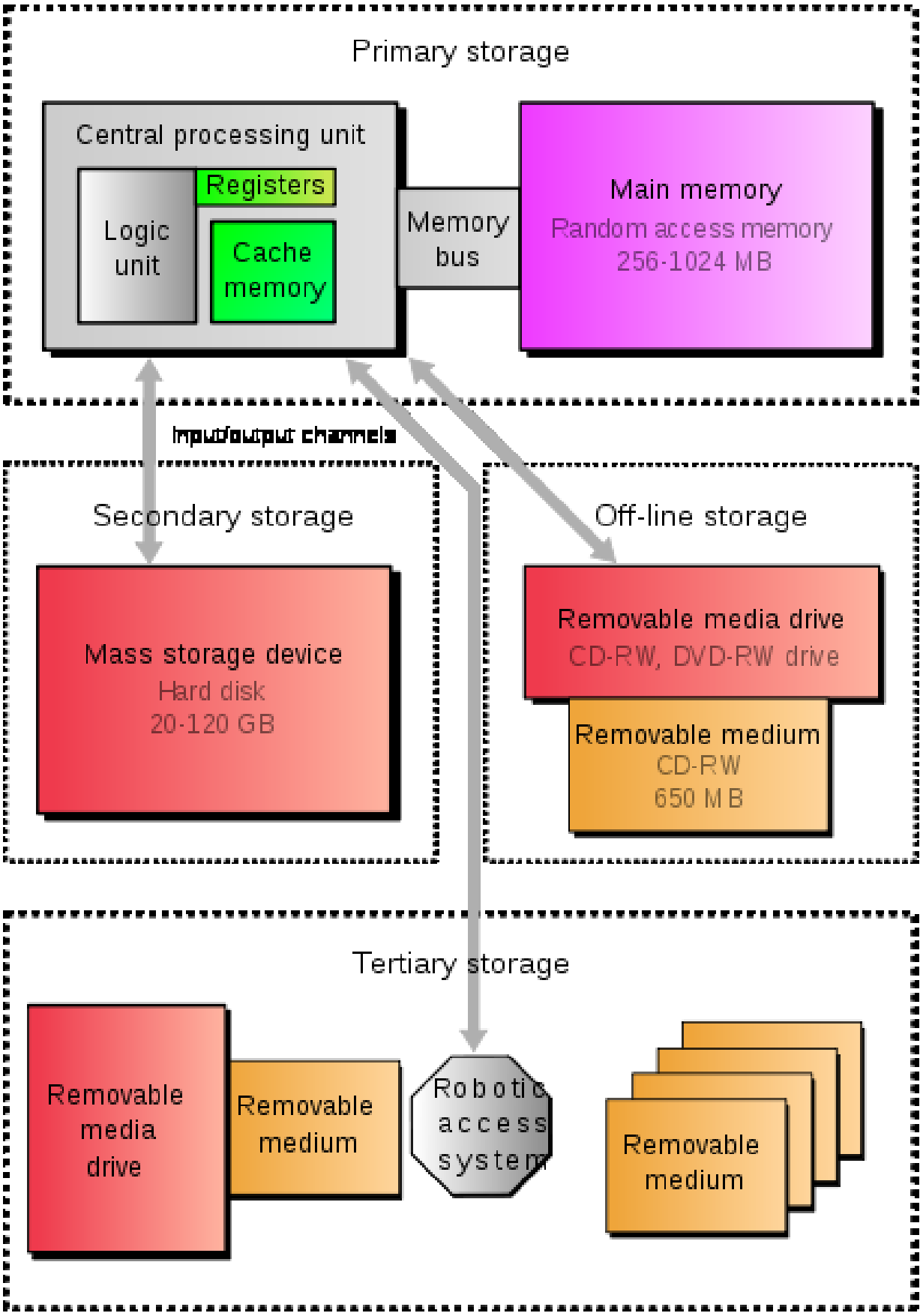


Figure 3.2: Computer architecture showing the interaction between the different levels of memory.

Chapter 4

The model diffusion equation – theoretical background

Overview

We consider analytical solutions to a two-dimensional diffusion problem. The reason for examining this particular problem are manifold: it is a minimal model that nonetheless has a small amount of complexity sufficient to warrant the use of a number of interesting numerical methods. Also, its analytical solution is rather interesting, as it requires the use of a hybrid Foruier-cosine series expansion. Finally, analytical solutions in this section will be used as benchmarks for future numerical simulation studies. Throughout the course, the problem considered in this section will be referred to as the **model diffusion equation**.

4.1 Boundary conditions – review

For this discussion, let

$$\frac{\partial C}{\partial t} = a(x,t)\frac{\partial^2 C}{\partial x^2} + b(x,t)\frac{\partial C}{\partial x} + c(x,t)C + d(x,t), \qquad x \in (0,L), \qquad t > 0,$$

be a parabolic partial differential equation in one space dimension, on $x \in (0,L)$, with smooth initial conditions

$$C(x, t=0) = C_{\text{init}}(x), \qquad x \in [0,L].$$

Then the following boundary conditions are possible.

1. **Dirichlet conditions** The function $C(x, t > 0)$ is specified on the boundaries:

$$\begin{aligned} C(0, t > 0) &= g_1(t), \\ C(L, t > 0) &= g_2(t). \end{aligned}$$

   If the functions $g_1 = g_2 = 0$, then we have **homogeneous Dirichlet conditions**:

$$\begin{aligned} C(0, t > 0) &= 0, \\ C(L, t > 0) &= 0. \end{aligned}$$

2. **Neumann conditions**: The **derivative** $C_x(x, t > 0)$ is specified on the boundaries:

$$\begin{aligned} C_x(0, t > 0) &= g_1(t), \\ C_x(L, t > 0) &= g_2(t). \end{aligned}$$

   If the functions $g_1 = g_2 = 0$, then we have **homogeneous Neumann conditions**, corresponding to **no flux through the boundaries**.

3. **Mixed conditions**: As the name suggests, this set is a mixture of Dirichlet and Neumann conditions:

$$\begin{aligned} \alpha_1 C_x(0, t > 0) + \alpha_2 C(0, t > 0) &= g_1(t), \\ \alpha_3 C_x(L, t > 0) + \alpha_4 C(L, t > 0) &= g_2(t). \end{aligned}$$

4. **Periodic boundary conditions**: The function $C(x, t > 0)$ has the same value on either boundary point:

$$C(0, t) = C(L, t), \qquad t > 0.$$

   In practice, these are not very realistic boundary conditions but they are used in numerical experiments because they are easy to implement. However, they can be used to mimic an infinite domain, if the periodic length L is made long enough.

4.2 The model diffusion equation

We are interested in solving the following partial differential equation (PDE) for diffusion, given here in non-dimensional form as follows:

$$\frac{\partial C}{\partial t} = \nabla^2 C + s(x, z), \qquad (x, z) \in \Omega, \tag{4.1a}$$

where

$$\Omega = (0, L_x) \times (0, 1), \tag{4.1b}$$

and $\nabla^2 = \partial_x^2 + \partial_z^2$ is the Laplacian. The partial differential equation is subject to the following boundary conditions:

$$\frac{\partial C}{\partial z} = 0, \qquad z = 0, \qquad z = 1, \tag{4.1c}$$

together with periodic boundary conditions in the x- and y-directions:

$$C(x = 0, z, t) = C(x = L_x, z, t). \tag{4.1d}$$

Finally, an initial condition is prescribed:

$$C(x, z, t = 0) = C_{\text{init}}(x, z), \qquad (x, z) \in \overline{\Omega}, \tag{4.1e}$$

where $C_{\text{init}}(x, z)$ is a continuous function. Here, the system of equations (6.1) is made non-dimensional on the channel depth L_z (herein set to unity), and the diffusive timescale $\tau = L_z^2/D$, where D is the diffusion coefficient.

4.3 Physical interpretation

Physically, Equation (6.1) is a model for diffusion of particles in the presence of a source. The amount of matter in the system changes over time, due to contributions from the source $s(x, y, z)$. There are no contributions from the boundary conditions. For, consider the following evolution equation for the total system mass

$$M = \int_\Omega \mathrm{d}^2x\, C(x, z, t).$$

We have

$$\begin{aligned} \frac{dM}{dt} &= \int_\Omega \mathrm{d}^2x\, \frac{\partial C}{\partial t}, \\ &= \int_\Omega \mathrm{d}^2x\, \left[\nabla^2 C + s(x, z)\right], \\ &= \int_\Omega \mathrm{d}^2x\, s(s, z) + \int_{\partial\Omega} \mathrm{d}A \boldsymbol{n} \cdot \nabla C, \end{aligned}$$

where $\partial\Omega$ is the boundary of the set Ω, $\mathrm{d}A$ is an element of area on the boundary, and $\boldsymbol{n}$ is the outward-pointing unit normal to $\partial\Omega$. We compute

$$\begin{aligned}\int_{\partial\Omega} \mathrm{d}A\boldsymbol{n}\cdot\nabla C &= \int_0^{L_x} \mathrm{d}x\,\frac{\partial C}{\partial z}\bigg|_{z=1} - \int_0^{L_x} \mathrm{d}x\,\frac{\partial C}{\partial z}\bigg|_{z=0} \\ &\quad + \int_0^{L_z} \mathrm{d}z\,\frac{\partial C}{\partial x}\bigg|_{x=L_x} - \int_0^{L_z} \mathrm{d}z\,\frac{\partial C}{\partial x}\bigg|_{x=0}.\end{aligned}$$

However, all of these terms cancel, either because of the no-flux boundary condition $(\partial C/\partial z)(z = 0, 1) = 0$, or because of periodicity, meaning that

$$\frac{dM}{dt} = L_x L_z \langle s\rangle, \qquad \langle s\rangle := \frac{1}{L_x L_z}\int_\Omega \mathrm{d}^2x\, s(s,z). \tag{4.2}$$

4.4 Decomposition

In view of the formula (4.2), it is sensible to split the solution into two parts:

$$C = \langle C\rangle(t) + C'(x,z,t), \qquad s = \langle s\rangle + s'(x,z).$$

By linearity,

$$\begin{aligned}\frac{\partial}{\partial t}\langle C\rangle &= \nabla^2\langle C\rangle + \langle s\rangle, \\ \frac{\partial C'}{\partial t} &= \nabla^2 C' + s'.\end{aligned}$$

Indeed, the solution to the mean contribution is known:

$$\langle C\rangle(t) = \langle C\rangle(t=0) + \langle s\rangle t = \langle C_{\mathrm{init}}\rangle + \langle s\rangle t.$$

while the PDE for the fluctuations inherits all the properties of the basic PDE (6.1), such that

$$\frac{\partial C'}{\partial t} = \nabla^2 C' + s'(x,z), \qquad (x,z)\in\Omega, \tag{4.3a}$$

subject to the following boundary conditions:

$$\frac{\partial C'}{\partial z} = 0, \qquad z = 0, \qquad z = 1, \tag{4.3b}$$

together with periodic boundary conditions in the x- and y-directions:

$$C'(x=0,z,t) = C'(x=L_x,z,t). \tag{4.3c}$$

Finally, an initial condition is prescribed:

$$C'(x,z,t=0) = C_{\text{init}}(x,z) - \langle C_{\text{init}} \rangle, \qquad (x,z) \in \overline{\Omega}, \tag{4.3d}$$

4.5 Analytical solution

We prove the following theorem:

Theorem 4.1 *Equation* (4.3) *has at least one smooth solution, namely*

$$C'(x,z,t) = \sum_{n=1}^{\infty} \sum_{i=-\infty}^{\infty} \left\{ \mathrm{e}^{-k_{in}^2 t} \left[a_{in}(0) - \frac{s_{in}}{k_{in}^2} \right] + \frac{s_{in}}{k_{in}^2} \right\} \mathrm{e}^{\mathrm{i}[(2\pi/L_x)ix]} \cos\left(\frac{n\pi z}{L_z}\right), \tag{4.4a}$$

where

$$a_{in}(0) = \frac{2}{L_x L_z} \int_0^{L_x} \int_0^{L_z} \mathrm{d}^2 x \, \mathrm{e}^{-\mathrm{i}[(2\pi/L_x)ix]} \cos\left(\frac{n\pi z}{L_z}\right) \left[C_0(x,z) - \langle C_0 \rangle \right], \tag{4.4b}$$

and where we have restored the definition of L_z for clarity's sake.

Proof: Take Equation (6.1a), multiply by $\cos(n\pi z/L_z)$ (with $n = 1, 2, \cdots$) and integrate with respect to z. The result is

$$\partial_t \int_0^{L_z} C' \cos\left(\frac{n\pi z}{L_z}\right) \mathrm{d}z = \partial_x^2 \int_0^{L_z} C' \cos\left(\frac{n\pi z}{L_z}\right) \mathrm{d}z + \int_0^{L_z} \left(\frac{\partial^2 C'}{\partial z^2}\right) \cos\left(\frac{n\pi z}{L_z}\right) \mathrm{d}z + \int_0^{L_z} s(x,z) \cos\left(\frac{n\pi z}{L_z}\right) \mathrm{d}z.$$

We call

$$\widehat{C}_n(x,t) := \frac{2}{L_z} \int_0^{L_z} C'(x,t) \cos\left(\frac{n\pi z}{L_z}\right) \mathrm{d}z, \qquad \widehat{s}_n(x) := \frac{2}{L_z} \int_0^{L_z} s(x,z) \cos\left(\frac{n\pi z}{L_z}\right) \mathrm{d}z;$$

hence, we have

$$\frac{\partial \widehat{C}_n}{\partial t} = \partial_x^2 \widehat{C}_n + \int_0^{L_z} \left(\frac{\partial^2 C'}{\partial z^2}\right) \cos\left(\frac{n\pi z}{L_z}\right) \mathrm{d}z + \widehat{s}_n(x,y). \tag{4.5}$$

Consider now the following (with $k_n = n\pi/L_z$):

$$\begin{aligned}
\int_0^{L_z} \left(\frac{\partial^2 C'}{\partial z^2}\right) \cos\left(\frac{n\pi z}{L_z}\right) \mathrm{d}z &= \int_0^{L_z} \left\{ \frac{\partial}{\partial z}\left[\cos\left(\frac{n\pi z}{L_z}\right)\frac{\partial C'}{\partial z}\right] + k_n \sin\left(\frac{n\pi z}{L_z}\right)\frac{\partial C'}{\partial z}\right\}\mathrm{d}z, \\
&= \left[\cos\left(\frac{n\pi z}{L_z}\right)\frac{\partial C'}{\partial z}\right]_{z=0}^{z=L_z} + k_n \int_0^{L_y} \sin\left(\frac{n\pi z}{L_z}\right)\frac{\partial C'}{\partial z}\,\mathrm{d}z, \\
&= 0 + k_n \int_0^{L_z} \sin\left(\frac{n\pi z}{L_z}\right)\frac{\partial C'}{\partial z}\,\mathrm{d}z, \\
&= k_n \int_0^{L_z} \left\{ \frac{\partial}{\partial z}\left[\sin\left(\frac{n\pi z}{L_z}\right) C'\right] - k_n \cos\left(\frac{n\pi z}{L_z}\right) C'\right\}\mathrm{d}z, \\
&= \left[\sin\left(\frac{n\pi z}{L_z}\right) C'\right]_{z=0}^{z=L_z} - k_n^2 \int_0^{L_z} \cos\left(\frac{n\pi z}{L_z}\right) C'(x,z,t)\,\mathrm{d}z.
\end{aligned}$$

Hence,

$$\frac{2}{L_z}\int_0^{L_z} \left(\frac{\partial^2 C'}{\partial z^2}\right) \cos\left(\frac{n\pi z}{L_z}\right) \mathrm{d}z = -k_n^2\,(2/L_z) \int_0^{L_z} \cos\left(\frac{n\pi z}{L_z}\right) C'(x,z,t)\,\mathrm{d}z = -k_n^2 \hat{C}_n.$$

Thus, Equation (6.1a) is transformed – via Equation (4.5) to

$$\frac{\partial \widehat{C}_n}{\partial t} + k_n^2 \widehat{C}_n = \partial_x^2 \widehat{C}_n + \widehat{s}_n(x). \tag{4.6}$$

However, this is now a standard diffusion problem in one (periodic) dimension, which can be solved by standard Fourier-series methods: we propose

$$\widehat{C}_n(x,t) = \sum_{i=-\infty}^{\infty} a_{in}(t) \mathrm{e}^{\mathrm{i}[(2\pi/L_x)ix]};$$

we also decompose $\widehat{s}_n(x,y)$ as

$$\widehat{s}_n(x) = \sum_{i=-\infty}^{\infty} s_{in} \mathrm{e}^{\mathrm{i}[(2\pi/L_x)ix]}.$$

Thus, the following amplitude equations are obtained:

$$\frac{da_{in}}{dt} = -k_{in}^2 a_{in} + s_{in}, \tag{4.7}$$

where

$$k_{in} = \left[\left(\frac{2\pi}{L_x}\right)^2 i^2 + \left(\frac{\pi}{L_z}\right)^2 n^2\right]^{1/2}.$$

Equation (4.7) has solution

$$a_{in}(t) = \begin{cases} \mathrm{e}^{-k_{in}^2 t}\left[a_{in}(0) - \frac{s_{in}}{k_{in}^2}\right] + \frac{s_{in}}{k_{ijn}^2}, & k_{in} \neq 0, \\ a_0(0) + s_0 t, & k_{in} = 0, \end{cases}$$

where the second case follows because $k_{in} = 0$ if and only if $i = n = 0$. However, this case is ruled out because $n \neq 0$. Thus, the solution for $\widehat{C}_n(x, y, t)$ is therefore

$$\widehat{C}_n(x,t) = \sum_{i=-\infty}^{\infty} \left\{ \mathrm{e}^{-k_{in}^2 t}\left[a_{in}(0) - \frac{s_{in}}{k_{in}^2}\right] + \frac{s_{in}}{k_{in}^2} \right\} \mathrm{e}^{\mathrm{i}[(2\pi/L_x)ix]}.$$

We now note that a cosine transform has been taken in the z-direction:

$$\widehat{C}_n(x,t) = \frac{2}{L_z}\int_0^{L_z} \cos\left(\frac{n\pi z}{L_z}\right) C'(x,z,t)\,\mathrm{d}z, \qquad n \neq 0.$$

However, since

$$\left\{ \cos\left(\frac{n\pi z}{L_z}\right) \right\}_{n=1}^{\infty}$$

is a basis for **mean-zero** continuous functions whose first derivative vanishes at $z = 0, L_z$, meaning that the cosine transform can be reversed:

$$C'(x,y,z,t) = \sum_{n=1}^{\infty} \widehat{C}_n(x,y,t) \cos\left(\frac{n\pi z}{L_z}\right).$$

Hence,

$$C'(x,z,t) = \sum_{n=1}^{\infty}\sum_{i=-\infty}^{\infty} \left\{ \mathrm{e}^{-k_{in}^2 t}\left[a_{in}(0) - \frac{s_{in}}{k_{in}^2}\right] + \frac{s_{in}}{k_{in}^2} \right\} \mathrm{e}^{\mathrm{i}[(2\pi/L_x)ix]} \cos\left(\frac{n\pi z}{L_z}\right).$$

Finally, it is of interest to determine the coefficients $a_{ijn}(0)$. We have

$$C_{\mathrm{init}}(x,y,z) - \langle C_{\mathrm{init}}\rangle = \sum_{n=1}^{\infty}\sum_{i=-\infty}^{\infty} a_{ij}(0)\mathrm{e}^{\mathrm{i}[(2\pi/L_x)ix]} \cos\left(\frac{n\pi z}{L_z}\right),$$

hence, by Fourier transformation,

$$a_{in}(0) = \frac{2}{L_x L_z}\int_0^{L_x}\int_0^{L_z} \mathrm{d}^2x\, \mathrm{e}^{-\mathrm{i}[(2\pi/L_x)ix]} \cos\left(\frac{n\pi z}{L_z}\right)\left[C_{\mathrm{init}}(x,z) - \langle C_{\mathrm{init}}\rangle\right].$$

(the factor of 2 comes from the cosine series). Having constructed a solution to Equation (4.3), it is also the case that this is the only such smooth solution:

Theorem 4.2 *Equation* (4.4) *is the unique smooth solution of Equation* (4.3)*.*

Exercise 4.1 *Prove Theorem* (4.2).

Chapter 5

The model Poisson problem

Overview

In this chapter we consider a simpler problem than the diffusion problem posed previously. It will be referred to throughout the course as the **model Poisson problem**.

5.1 The model Poisson problem

Consider the following sample problem (Poisson equation):

$$\nabla^2 C_0 + s(x,z) = 0, \qquad (x,z) \in \Omega, \tag{5.1a}$$

where

$$\Omega = (0, L_x) \times (0,1), \tag{5.1b}$$

with boundary conditions

$$\frac{\partial C_0}{\partial z} = 0, \qquad z = 0, \qquad z = 1, \tag{5.1c}$$

and

$$C_0(x = 0, z, t) = C_0(x = L_x, z, t). \tag{5.1d}$$

5.2 Solvability condition and explicit solution

Consider Equation (5.1a). Integrate both sides over x and z and apply the boundary conditions on C. The result is

$$0 = \int_0^{L_x} \mathrm{d}x \int_0^{L_z} \mathrm{d}z\, s(x,z).$$

Thus, in order to get a self-consistent solutin, we require that the source should have zero mean:

$$\langle s \rangle := \frac{1}{L_x L_z} \int_0^{L_x} \mathrm{d}x \int_0^{L_z} \mathrm{d}z\, s(x,z) = 0.$$

This is referred to as the **solvability condition**.

Assuming that Equation (5.1) satisfies the solvability condition, a solution is available through a Fourier-cosine series:

$$C_0(x,y) = \sum_{n=1}^{\infty} \sum_{i=-\infty}^{\infty} \frac{s_{in}}{k_{in}^2} \mathrm{e}^{\mathrm{i}[(2\pi/L_x)ix]} \cos\left(\frac{n\pi z}{L_z}\right), \tag{5.2}$$

where k_{in} and s_{in} are defined as in Chapter 4.

5.3 Relation between model diffusion and Poisson problems

It is clear from Equation (5.2) that

$$\lim_{t\to\infty} C'(x,z,t) = C_0(x,z),$$

where $C'(x,z,t)$ (LHS) is the fluctuating part of the solution of the model diffusion problem, and $C_0(x,y)$ (RHS) here denotes the solution of the model Poisson problem. Thus, the following theorem is shown:

Theorem 5.1 *Let $\langle s \rangle = 0$ in the model diffusion equation. Then the solution $C(x,z,t)$ of the model diffusion equation – given smooth initial data – converges to the solution of the model Poisson problem, as $t \to \infty$:*

$$\lim_{t\to\infty} C(x,z,t) = C_0(z,z), \qquad \nabla^2 C_0(x,y) + s(x,y) = 0.$$

Exercise 5.1 *Prove Theorem 5.1 using a second approach: show first that*

$$\frac{\partial}{\partial t}\left(C - C_0\right) = \nabla^2\left(C - C_0\right),$$

and hence show that

$$\lim_{t\to\infty} C(x, y, t) = C_0(x, y). \tag{5.3}$$

Non-uniqueness of solutions

Consider again the model Poisson problem with solution (5.2). It is clear that $C_0 + \mathrm{Const.}$ is also a solution, since $\nabla^2(\mathrm{Const.}) = 0$, and $C_0 + \mathrm{Const.}$ also satisfies the boundary conditions (hybrid periodic–Neumann). The solution (5.2) is therefore not unique. This is because the operator ∇^2, equipped with the hybrid periodic–Neumann boundary conditions has a non-trivial kernel – the set of all constant functions is a one-dimensional vector subspace and is the non-trivial kernel of the PDE.

Exercise 5.2 *What happens to the kernel of ∇^2 if the boundary conditions are modified to be a mixture of periodic BCs in the x-direction and homogeneous Dirichlet conditions in the z-direction?*

Pedagogical link between the model diffusion and Poisson problems

Throughout the rest of the course, work will be split up in the following way

- I will show you how to solve the model Poisson problem numerically.
- You will then take these techniques and solve the more difficult model diffusion equation.

Chapter 6

Model diffusion equation – Numerical setup

Overview

In this chapter we consider numerical solutions of the model diffusion equation. We introduce centred differencing in space as a way of approximating the Laplace operator numerically. Also, Crank–Nicholson temporal discretization is introduced as a way of discretizing the temporal derivative $\partial/\partial t$. Crank–Nicholson is a so-called **implicit method**, which means that a certain equation must be inverted in order to evolve the numerical solution forward in time, stepping from one time step to the next. For this reason, Jacobi iteration is introduced as a method for solving such implicit (linear) equations.

6.1 The model

We are interested in solving the PDE from Chapter 4, recalled here to be

$$\frac{\partial C}{\partial t} = \nabla^2 C + s(x,z), \qquad (x,z) \in \Omega, \tag{6.1a}$$

where

$$\Omega = (0, L_x) \times (0,1), \tag{6.1b}$$

and $\nabla^2 = \partial_x^2 + \partial_z^2$ is the Laplacian. The partial differential equation is subject to the following boundary conditions:

$$\frac{\partial C}{\partial z} = 0, \qquad z = 0, \qquad z = 1, \tag{6.1c}$$

together with periodic boundary conditions in the x- and y-directions:

$$C(x=0,z,t) = C(x=L_x,z,t). \tag{6.1d}$$

Finally, an initial condition is prescribed:

$$C(x,z,t=0) = C_{\text{init}}(x,z), \qquad (x,z) \in \overline{\Omega}, \tag{6.1e}$$

where $C_{\text{init}}(x,z)$ is a continuous function. Here, the system of equations (6.1) is made non-dimensional on the channel depth L_z (herein set to unity), and the diffusive timescale $\tau = L_z^2/D$, where D is the diffusion coefficient.

6.2 The discretization

We discretize the PDE and compute its approximate numerical solution on a discrete grid:

$$\begin{aligned} x_i &= (i-1)\Delta x, \qquad i = 1, \cdots n_x, \\ y_j &= (j-1)\Delta y, \qquad j = 1, \cdots n_y, \end{aligned}$$

such that

$$(n_x - 1)\Delta x = L_x, \qquad \Delta x = L_x/(n_x - 1),$$

and similarly, $\Delta y = L_y/(n_y - 1)$. The PDE is also discretized in time, such that the solution is only available at discrete points in time $t_n = n\Delta t$, with $n = 0, 1, \cdots$. The solution at t_n and $\boldsymbol{x} = (i\Delta x, y\Delta j)$ is written as C_{ij}^n. The diffusion operatior in the PDE (6.1) is approximated by **centred differences**:

$$\left(\nabla^2 C\right)_{ij} \approx \frac{C_{i+1,j} + C_{i-1,j} - 2C_{ij}}{\Delta x^2} + \frac{C_{i,j+1} + C_{i,j-1} - 2C_{ij}}{\Delta y^2} := \mathcal{D}(C_{ij})$$
$$i = 2, 2, \cdots, n_x - 1, \qquad j = 2, 2, \cdots, n_y - 1.$$

> **Common Programming Error:**
>
> Extending the index ranges on i and j down to $i = 1, j = 1$ and up to $i = n_x, j = n_y$. This is not allowed, since these are boundary points, where the PDE does not apply. Instead, boundary conditions apply to these sets of points.

The discretization in time is done using a **Crank–Nicholson** scheme:

$$\frac{C_{ij}^{n+1} - C_{ij}^{n}}{\Delta t} = \tfrac{1}{2}\left[\mathcal{D}(C_{ij}^{n}) + \mathcal{D}(C_{ij}^{n+1})\right] + s_{ij}, \qquad i = 2, 2, \cdots, n_x - 1, \qquad j = 2, 2, \cdots, n_y - 1.$$

Re-arrange:

$$\left[1 - \tfrac{1}{2}\Delta t\mathcal{D}\right]\left(C_{ij}^{n+1}\right) = \left[1 + \tfrac{1}{2}\Delta t\mathcal{D}\right]\left(C_{ij}^{n}\right) + \Delta t s_{ij}, \qquad i = 2, 2, \cdots, n_x - 1, \qquad j = 2, 2, \cdots, n_y - 1. \quad (6.2)$$

On the left-hand side, the quantity $\left[1 - \frac{1}{2}\Delta t\mathcal{D}\right]$ is in fact a matrix operator, and the solution is available only in **implicit** form: an inversion needs to be performed to extract C_{ij}^{n+1} from this implicit equation:

$$C_{ij}^{n+1} = \left[1 - \tfrac{1}{2}\Delta t\mathcal{D}\right]^{-1}\left\{\left[1 + \tfrac{1}{2}\Delta t\mathcal{D}\right]\left(C_{ij}^{n}\right) + \Delta t s_{ij}\right\}. \quad (6.3)$$

The implicit equation (6.2) is written out in more detail now:

$$(1 + a_x + a_y)\, C_{ij}^{n+1} - \tfrac{1}{2}a_x\left(C_{i+1,j}^{n+1} + C_{i-1,j}^{n+1}\right) + \tfrac{1}{2}a_y\left(C_{i,j+1}^{n+1} + C_{i,j-1}^{n+1}\right) = \left[1 + \tfrac{1}{2}\Delta t\mathcal{D}\right]\left(C_{ij}^{n}\right) + s_{ij} := \mathrm{RHS}_{ij}^{n},$$

where $a_x = \Delta t/\Delta x^2$ and $a_z = \Delta t/\Delta z^2$. Tidy up:

$$(1 + a_x + a_y)\, C_{ij}^{n+1} - \tfrac{1}{2}a_x\left(C_{i+1,j}^{n+1} + C_{i-1,j}^{n+1}\right) + \tfrac{1}{2}a_y\left(C_{i,j+1}^{n+1} + C_{i,j-1}^{n+1}\right) = \mathrm{RHS}_{ij}^{n}. \quad (6.4)$$

This is an implicit equation for C_{ij}^{n+1} that we must now endavour to solve.

Exercise 6.1 *Investigate the reaons why a Crank–Nicholson treatment of the diffusion operator is preferred.*

6.3 Jacobi method

The focus of this course is on the use of **iterative methods** to solve problems such as Equation (6.3) or equivalently, Equation (6.4). The idea is to make an initial guess for the solution, plug this into some algorithm for refining the guess, and continue until this iterative procedure converges.

The simplest and most naive iterative method os the so-called **Jacobi method**. Let $v \equiv C^{n+1}$ be the array to be found by solving Equation (6.3):

$$(1 + a_x + a_y)\, v_{ij} - \tfrac{1}{2}a_x\left(v_{i+1,j} + v_{i-1,j}\right) + \tfrac{1}{2}a_y\left(v_{i,j+1} + v_{i,j-1}\right) = \mathrm{RHS}_{ij}.$$

This can be re-arranged simply as

$$v_{ij} = \frac{\frac{1}{2}a_x\left(v_{i+1,j} + v_{i-1,j}\right) + \frac{1}{2}a_y\left(v_{i,j+1} + v_{i,j-1}\right) + \mathrm{RHS}_{ij}}{1 + a_x + a_y}. \tag{6.5}$$

The idea of the Jacobi method is to take a guess for v, say v^N, and to create a new guess v_{N+1} via the formula

$$v_{ij}^{N+1} = \frac{\frac{1}{2}a_x\left(v_{i+1,j}^{N} + v_{i-1,j}^{N}\right) + \frac{1}{2}a_y\left(v_{i,j+1}^{N} + v_{i,j-1}^{N}\right) + \mathrm{RHS}_{ij}}{1 + a_x + a_y}. \tag{6.6}$$

If this iterative scheme converges, then $\lim_{N\to\infty} v^N = \lim_{N\to\infty} v^{N+1}$, and the approximate solutions v^N and v^{N+1} can be replaced in Equation (6.6) with some identical array v^*, thereby forcing Equation (6.6) to be identical to Equation (6.5).

Common Programming Error:

Mixing up the iteration level N and the time-step levle n in an application of an iterative method to an evolusionary PDE.

6.4 Boundary conditions

The idea to solve the PDE (6.1a) is to do implement a Crank–Nicholson-centred difference scheme at interior points. Inversion of the resulting implicit problem is then achieved by the Jacobi method. However, this approach can only be used at **interior points**

$$i = 2, 2, \cdots, n_x - 1, \qquad j = 2, 2, \cdots, n_y - 1.$$

At boundary points, the boundary conditions are enforced: $\partial C/\partial z = 0$ at $z = 0, 1$, and periodic boundary conditions in the x-direction. These are implemented numerically in a straightforward fashion. The Neuman conditions at $z = 0, L_z$ are implemented as

$$C_{i,j=1} = C_{i,j=2}, \qquad C_{i,j=n_y} = C_{i,j=n_y-1},$$

while the periodicity conditions at $x = 0, L_x$ are implemented as follows

- $i = 1$: $C(i-1, j) = C(n_x - 1, j)$,
- $i = n_x$: $C(i+1, j) = C(2, j)$.

Thus, the points $i = 1$ and $i = n_x$ are identified.

6.5 The algorithm

We can now assemble an algorithm to solve Equation (6.1a) numerically:

1. Set up a discretization scheme with Δx, Δz, and Δt defined by the user. Also, prescribe an initial condition $C_{\text{init}}(x, z)$ and a source function $s(x, z)$.
2. Obtain $C_{ij}^{n=1}$ from C_{ij}^{0} at interior points using centred differences, the Crank–Nicholson temporal discretization, and Jacobi iteration.
3. Implement many iterations of the Jacobi method, until the method has converged to some user-defined tolerance.
4. Implement the boundary conditions on $C^{n=1}$.
5. Repeat steps 2–4 for the desired number of timesteps.

Chapter 7

The model Poisson problem – numerical setup

Overview

In this chapter we consider numerical solutions to the model Poisson problem.

7.1 The code

A numerical method for solving this problem is implemented below.

```
function [xx,yy,C,C_true,res_it]=test_poisson_jacobi()

% Numerical method to solve
% [D_{xx}+D_{yy}]C=s(x,y),
% subject to periodic boundary conditions in the x-direction,
% and Neuman boundary conditions at y=0 and y=L_y.

aspect_ratio=2;

Ny=101;
Nx=aspect_ratio*(Ny-1)+1;

iteration_max=5000;

Ly=1.d0;
Lx=aspect_ratio*Ly;
dx=Lx/(Nx-1);
dy=Ly/(Ny-1);
```

```
19
20 kx0=2*pi/Lx;
21 ky0=pi/Ly;
22 A0=10;
23
24 dx2=dx*dx;
25 dy2=dy*dy;
26
27 f_source=zeros(Nx,Ny);
28
29 % Initialise source
30
31 kx=kx0;
32 ky=3*ky0;
33
34 for i=1:Nx
35     for j=1:Ny
36       x_val=(i-1)*dx;
37       y_val=(j-1)*dy;
38       f_source(i,j)=A0*cos(kx*x_val)*cos(ky*y_val);
39     end
40 end
41
42 % Compute analytic solution ************************************************
43
44
45 xx=0*(1:Nx);
46 yy=0*(1:Ny);
47 C_true=zeros(Nx,Ny);
48
49 for i=1:Nx
50     for j=1:Ny
51       xx(i)=(i-1)*dx;
52       yy(j)=(j-1)*dy;
53
54       C_true(i,j)=(-A0/(kx*kx+ky*ky))*cos(kx*xx(i))*cos(ky*yy(j));
55
56     end
57 end
58
59 % Iteration step ***********************************************************
60 % Initial guess for C:
61 C=zeros(Nx,Ny);
62
63 res_it=0*(1:iteration_max);
```

```
64
65 for iteration=1:iteration_max
66
67     C_old=C;
68
69     for i=1:Nx
70
71         % Periodic BCs here.
72         if(i==1)
73             im1=Nx-1;
74         else
75             im1=i-1;
76         end
77
78         if(i==Nx)
79             ip1=2;
80         else
81             ip1=i+1;
82         end
83
84         for j=2:Ny-1
85
86             diagonal=(2.d0/dx2)+(2.d0/dy2);
87             tempval=(1.d0/dx2)*(C_old(ip1,j)+C_old(im1,j))+(1.d0/dy2)*(C_old(i,j
                   +1)+C_old(i,j-1))-f_source(i,j);
88             C(i,j)=tempval/diagonal;
89
90         end
91     end
92
93     % Implement Dirichlet conditions at y=0,y=L_y.
94     C(:,1)=C(:,2);
95     C(:,Ny)=C(:,Ny-1);
96
97     res_it(iteration)=max(max(abs(C-C_old)));
98
99 end
100
101 end
```

codes/poisson_matlab/test_poisson_jacobi.m

7.2 Now you try...

Exercise 7.1 *Write a Matlab code to solve the full diffusion equation* (6.1a)*. Use the model Poisson problem as a starting-point. Also, use the* **Jacobi iteration method** *in the Crank–Nicholson step. Take* $C_0(x, z) = 0$*, and let* $s(x, y)$ *be a simple combination of a small number of sines and cosines, with wavelengths that fit inside the box* $(0, L_x) \times (0, L_z)$*. In this way, validate your Matlab code with respect to the analytical solution obtained in Chapter 4.*

Chapter 8

Jacobi iteration – convergence

Overview

The idea of this chapter is to take a generic linear problem $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$, and to formulate a sufficient condition on $\mathbf{A}$ that guarantees the success of the Jacobi iterative method. It turns out that this sufficient condition is something called **diagonal dominance**, which means that the diagonal elements of $\mathbf{A}$ should be large (in some sense) compared to the off-diagonal ones.

8.1 Generic discussion

Consider the Jacobi scheme for solving

$$\mathbf{A}\boldsymbol{v} = \boldsymbol{b}, \qquad (\mathbf{A})_{ij} = a_{ij} \in \mathbb{R}.$$

The idea is to write $\mathbf{A} = \mathbf{D} + \mathbf{R}$, where

$$\mathbf{D} = \operatorname{diag}\left(a_{11}, a_{22}, \cdots, a_{nn}\right), \qquad n \in \mathbb{N},$$

and where

$$\mathbf{R} = \mathbf{A} - \mathbf{D}.$$

Then, the iterations $\boldsymbol{v}^N, \boldsymbol{v}^{N+1}$ that generate approximate solutions are obtained as follows:

$$\mathbf{D}\boldsymbol{v}^{N+1} = -\mathbf{R}\boldsymbol{v}^N + \boldsymbol{b},$$

for a given starting-guess $\boldsymbol{v}^0$. Assume that

$$\lim_{N\to\infty} = \boldsymbol{v}^*.$$

Thus,

$$\begin{aligned} \mathbf{D}\boldsymbol{v}^{N+1} &= -\mathbf{R}\boldsymbol{v}^N + \boldsymbol{b}, \\ \mathbf{D}\boldsymbol{v}^* &= -\mathbf{R}\boldsymbol{v}^* + \boldsymbol{b}. \end{aligned}$$

Subtract:

$$\mathbf{D}\left(\boldsymbol{v}^{N+1} - \boldsymbol{v}^*\right) = -\mathbf{R}\left(\boldsymbol{v}^N - \boldsymbol{v}^*\right),$$

or

$$\mathbf{D}\boldsymbol{r}^{N+1} = -\mathbf{R}\boldsymbol{r}^N, \tag{8.1}$$

where $\boldsymbol{r}^N = \boldsymbol{v}^N - \boldsymbol{v}^*$ is the **residual vector** at level N. Take L^2 vector norms on both sides:

$$\|\boldsymbol{r}^{N+1}\| = \|\mathbf{D}^{-1}\mathbf{R}\boldsymbol{r}^N\|_2.$$

Now, use the L^2 operator norm (Appendix A):

$$\|\boldsymbol{r}^{N+1}\|_2 \leq \|\mathbf{D}^{-1}\mathbf{R}\|_2\|\boldsymbol{r}^N\|_2.$$

Telescope this result:

$$\|\boldsymbol{r}^N\|_2 \leq \left(\|\mathbf{D}^{-1}\mathbf{R}\|_2\right)^N \|\boldsymbol{r}^0\|_2.$$

By requiring that $\|\mathbf{D}^{-1}\mathbf{R}\|_2 < 1$, we obtain

$$\lim_{N\to\infty} \|\boldsymbol{r}^N\|_2 = 0,$$

hence

$$\lim_{N\to\infty} \boldsymbol{r}^N = 0,$$

hence

$$\lim_{N\to\infty} \boldsymbol{v}^n = \boldsymbol{v}^*.$$

Thus, we have shown the following theorem:

Theorem 8.1 *A sufficient condition for the convergence of the Jacobi iteration algorithm*

$$\mathbf{D}\boldsymbol{v}^{N+1} = -\mathbf{R}\boldsymbol{v}^{N} + \boldsymbol{b}, \qquad \boldsymbol{v}^{N=0} = \boldsymbol{v}_0,$$

is the following bound on the L^2 operator norm:

$$\|\mathbf{D}^{-1}\mathbf{R}\|_2 < 1. \tag{8.2}$$

For systems with entries on the diagonal that are relatively large (in absolte-value terms) compared to entries off the diagonal, this constraint is usually satisfied, and the Jacobi iteration converges. However, this is a relatively vague criterion, which is of limited use. In addition, the L^2 operator norm is difficult to compute numerically, so in practice it is not known *a priori* – using Theorem (8.1) alone – whether the Jacobi method will converge. For that reason, we need a more rigorous notion of diagonal dominance.

8.2 Diagonal dominance

Definition 8.1 (L^∞ matrix norm) *Let $\mathrm{M} \in \mathbb{R}^{n\times n}$. Then*

$$\|\mathbf{M}\|_\infty = \sup_{\|\boldsymbol{x}\|_\infty = 1} \|\mathbf{M}\boldsymbol{x}\|_\infty,$$

where $\|\boldsymbol{x}\|_\infty$ denotes the ordinary L^∞ norm for vectors: for $\boldsymbol{x} = (x_1, \cdots, x_n) \in \mathbb{R}^n$,

$$\|\boldsymbol{x}\|_\infty = \max_i |x_i|.$$

Lemma 8.1 (Consistency of the L^∞ norm) *Let $\mathbf{M}_1$ and $\mathbf{M}_2$ be square matrices in $\mathbb{R}^{n\times n}$. Then*

$$\|\mathbf{M}_1\mathbf{M}_2\|_\infty \leq \|\mathbf{M}_1\|_\infty \|\mathbf{M}_2\|_\infty.$$

Proof: It suffices to consider the case with $\|\mathbf{M}_1\mathbf{M}_2\|_\infty \neq 0$.

$$\begin{aligned}
\|\mathbf{M}_1\mathbf{M}_2\|_\infty &= \sup_{\|\boldsymbol{x}\|_\infty=1} \|\mathbf{M}_1\mathbf{M}_2\boldsymbol{x}\|_\infty, \\
&= \|\mathbf{M}_1\mathbf{M}_2\boldsymbol{x}_0\|_\infty, \\
&= \|\mathbf{M}_1\boldsymbol{y}\|_\infty, \qquad \boldsymbol{y} = \mathbf{M}_2\boldsymbol{x}_0 \neq 0, \\
&= \left(\frac{\|\mathbf{M}_1\boldsymbol{y}\|_\infty}{\|\boldsymbol{y}\|_\infty}\right)\|\boldsymbol{y}\|_\infty, \\
&\leq \left[\sup_{\boldsymbol{y}\neq 0}\left(\frac{\|\mathbf{M}_1\boldsymbol{y}\|_\infty}{\|\boldsymbol{y}\|_\infty}\right)\right]\|\boldsymbol{y}\|_\infty, \\
&= \|\mathbf{M}_1\|_\infty\|\boldsymbol{y}\|_\infty, \\
&= \|\mathbf{M}_1\|_\infty\|\mathbf{M}_2\boldsymbol{x}_0\|_\infty, \\
&\leq \|\mathbf{M}_1\|_\infty\|\mathbf{M}_2\|_\infty.
\end{aligned}$$

Definition 8.2 *Let $\mathbf{M} \in \mathbb{R}^{n\times n}$. The spectral radius $\rho(\mathbf{M})$ refers to that $\mathbf{M}$-eigenvalue with maximal absolute value:*

$$\rho(\mathbf{M}) = \max_i \left(|\lambda_i|\right), \qquad \mathbf{M}\boldsymbol{x}_i = \lambda_i\boldsymbol{x}_i.$$

Theorem 8.2 (Bound on the spectral radius) *Let $\mathbf{M} \in \mathbb{R}^{n\times n}$. Then*

$$\rho(\mathbf{M}) \leq \|\mathbf{M}\|_\infty.$$

Proof: Let $\mathbf{M}\boldsymbol{x} = \lambda\boldsymbol{x}$, with $\boldsymbol{x} \neq 0$. Let

$$\mathbf{X} = \begin{pmatrix} | & & | \\ \boldsymbol{x} & \cdots & \boldsymbol{x} \\ | & & | \end{pmatrix}.$$

Thus,

$$\mathbf{MX} = \lambda\mathbf{X}.$$

Take L^∞ norms on both sides:

$$|\lambda|\|\mathbf{X}\|_\infty = \|\mathbf{M}\boldsymbol{x}\|_\infty \leq \|\mathbf{M}\|_\infty\|\mathbf{X}\|_\infty,$$

hence

$$|\lambda| \leq \|\mathbf{M}\|_\infty,$$

for any eigenvalue λ, and the result is shown:

$$\rho(\mathbf{M}) \leq \|\mathbf{M}\|_\infty.$$

Now, by this stage, I am tired of proving theorems, so I shall simply state this last and crucial theorem:

Theorem 8.3 *Let $\mathbf{M} \in \mathbb{R}^{n\times n}$. Then*

$$\|\mathbf{M}\|_\infty = \max_i \left(\sum_{k=1}^{n} |m_{ik}| \right). \tag{8.3}$$

Note the sum over columns!

Putting it all together

We now apply these results to $\mathbf{M} = \mathbf{D}^{-1}\mathbf{R}$. In view of Theorem 8.2 and Theorem 8.3, we have

$$\rho\left(\mathbf{D}^{-1}\mathbf{R}\right) \leq \|\mathbf{D}^{-1}\mathbf{R}\|_\infty = \max_i \left(\sum_{k=1}^{n} |(\mathbf{D}^{-1}\mathbf{R})_{ik}| \right)$$

But

$$\mathbf{D}^{-1}\mathbf{R} = \begin{pmatrix} 0 & \frac{a_{12}}{a_{11}} & \cdots & \frac{a_{1,n-1}}{a_{11}} & \frac{a_{1n}}{a_{11}} \\ \vdots & & & & \\ \frac{a_{n1}}{a_{nn}} & \cdots & & \frac{a_{n,n-1}}{a_{nn}} & 0 \end{pmatrix},$$

hence

$$\rho\left(\mathbf{D}^{-1}\mathbf{R}\right) \leq \|\mathbf{D}^{-1}\mathbf{R}\|_\infty = \max_i \left(\frac{1}{|a_{ii}|} \sum_{\substack{k=1 \\ k\neq i}}^{n} |a_{ik}| \right).$$

This motivates a definition:

Definition 8.3 *A matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ is diagonally dominant if*

$$\frac{1}{|a_{ii}|} \sum_{\substack{k=1 \\ k\neq i}}^{n} |a_{ik}| < 1,$$

for each $i = 1, 2, \cdots, n$.

Along the way, we have established the following facts for a diagonally-dominant matrix:

Theorem 8.4 *Let $\mathbf{A} \in \mathbb{R}^{n\times n}$ be diagonally dominant. Then*

$$\rho\left(\mathbf{D}^{-1}\mathbf{R}\right) \leq \|\mathbf{D}^{-1}\mathbf{R}\|_\infty = \max_i \left(\frac{1}{|a_{ii}|} \sum_{\substack{k=1 \\ k\neq i}}^{n} |a_{ik}| \right) < 1,$$

where $\mathbf{D}$ and $\mathbf{R}$ have their usual meanings.

We now consider a final theorem:

Theorem 8.5 *Let $\mathbf{A}\boldsymbol{v} = \boldsymbol{b}$ be a linear problem, where $\mathbf{A} \in \mathbb{R}^{n\times n}$ is diagonally dominant. Then the Jacobi iteration method converges.*

Proof: Start with the definition of the Jacobi residuals, Equation (8.1) or, equivalently,

$$\boldsymbol{r}^{N+1} = -\mathbf{D}^{-1}\mathbf{R}\boldsymbol{r}^{N}, \tag{8.4}$$

Telescope the result:

$$\boldsymbol{r}^{N} = (-1)^{N}\left(\mathbf{D}^{-1}\mathbf{R}\right)^{N}\boldsymbol{r}^{0}.$$

Write

$$\mathbf{D}^{-1}\mathbf{R} = \mathbf{P}^{-1}\mathbf{J}\mathbf{P},$$

where $\mathbf{J}$ is the Jordan normal form associated with $\mathbf{D}^{-1}\mathbf{R}$. We have

$$\begin{aligned}\boldsymbol{r}^{N} &= (-1)^{N}\left(-\mathbf{D}^{-1}\mathbf{R}\right)^{N}\boldsymbol{r}^{0},\\ &= (-1)^{N}\left(\mathbf{P}^{-1}\mathbf{J}\mathbf{P}\right)\boldsymbol{r}^{0},\\ &= (-1)^{N}\left(\mathbf{P}^{-1}\mathbf{J}^{N}\mathbf{P}\right)\boldsymbol{r}^{0}.\end{aligned}$$

We now use the fact that $\rho(\mathbf{D}^{-1}\mathbf{R}) \leq \|\mathbf{D}^{-1}\mathbf{R}\|_{\infty} < 1$ as in the hypothesis of the theorem. Thus, all the eigenvalues have modulus less than one. Hence, each block in the Jordan matrix, raised to the N^{th} power, tends to zero as $N \to \infty$. It follows that

$$\lim_{N\to\infty}\mathbf{J}^{N} = 0. \tag{8.5}$$

Therefore, finally,

$$\lim_{N\to\infty}\boldsymbol{r}^{N} = 0.$$

Exercise 8.1 *Familiarize yourself with the Jordan decomposition (Appendix A), and satisfy yourself that Equation* (8.5) *is true.*

8.3 Operation count

An elementary (non-iterative) method of solving linear problems is Gaussian elimination. The operation count of Gaussian elimination is $O(n^3)$, meaning that the number of operations (addition, multiplication etc.) required to invert the matrix is proprtional to the cube of the size of the matrix. This can be regarded as a relatively good performance result, since it compares very favourably with the operation count of determinant-type calculations, the latter being another candidate method for matrix inversion. However, for massive calculations (e.g. $n \sim 10^6$), even the relatively good performance of Gaussian elimination ($O(n^3)$) is not satisfactory. For such large calculations, iterative methods such as the Jacobi scheme are preferred; clearly in such iterative methods, the count is $O(n_c n^2)$, where n_c is the number of iterations required for the method to converge, with $n_c \ll n$ for n large.

Chapter 9

Successive over-relaxation

Overview

Recall theorem (8.4) in Chapter 8: given a diagonally-dominant problem $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$, the Jacobi iteration method will converge. In this module, we are always working with such systems. Thus, the Jacobi method will always work for us. However, its convergence is quite poor. In other words, a relatively large number of iterations is required in order to obtain a sufficiently converged solution. In this section we outline a new method. Superficially, it is a straightforward extension of the Jacobi method; however, on deeper reflection, the improved method represents a conceptual leap. This is the method of **successive over-relaxation** (SOR).

9.1 The idea

Start with the generic problem

$$\mathbf{A}\boldsymbol{x} = \boldsymbol{b}.$$

Recall the Jacobi solution:

$$\mathbf{D}\boldsymbol{v}^{N+1} = -\mathbf{R}\boldsymbol{v}^{N} + \boldsymbol{b}.$$

In index notation,

$$v_i^{N+1} = -\frac{1}{a_{ii}} \sum_{j=1}^{n} R_{ik} v_k^N + b_i. \tag{9.1}$$

The idea behind SOR is to retrospectively improve the 'old guess' $\boldsymbol{v}^N$ that goes into formulating the 'new guess'. If the 'old guess' can be retrospectively improved, then this makes the new guess even better. To do this, the right-hand side of the Jacobi equation (9.1) is updated with just-recently-created values of $\boldsymbol{v}^{N+1}$. Where this is not possible, the old values of $\boldsymbol{v}^N$ are used. The result is the

following iterative scheme:

$$v_i^{N+1} = -\frac{1}{a_{ii}} \sum_{k=1}^{i-1} R_{ik} v_k^{N+1} - \frac{1}{a_{ii}} \sum_{k=i}^{n} R_{ik} v_k^N + \frac{b_i}{a_{ii}}. \tag{9.2}$$

But $R_{ii} = 0$, and $R_{ij} = a_{ij}$ otherwise. Hence, Equation (9.2) can be replaced by

$$v_i^{N+1} = \frac{1}{a_{ii}} \left[b_i - \sum_{k=1}^{i-1} a_{ik} v_k^{N+1} - \sum_{k=i+1}^{n} a_{ik} v_k^N \right]. \tag{9.3}$$

Equation (9.3) is not yet optimal (however, it is already the Gauss–Seidel method for solving a linear system). Instead, we introduce an extra degree of freedom, which allows us to weight how much or how little retrospective improvement of the old guess is implemented in the $(N+1)^{\text{th}}$ iteration step. This is done by a simple modification of Equation (9.3):

$$v_i^{N+1} = (1-\omega)\, a_{ii} + \frac{\omega}{a_{ii}} \left[b_i - \sum_{k=1}^{i-1} a_{ik} v_k^{N+1} - \sum_{k=i+1}^{n} a_{ik} v_k^N \right] \tag{9.4a}$$

The factor ω is restricted to the range

$$0 < \omega < 2; \tag{9.4b}$$

this preserves the diagonal-dominance of the system and hence ensures convergence. The exact choice of ω is made by trial-and-error in order to speed up convergence.

9.2 Implementation – model Poisson problem

In the example below, I started with the problem in Chapter 7 and the associated Jacobi solver. I have replaced the Jacobi solver with an SOR solver – minimal changes are required. That said, the concept is quite different.

```
function [xx,yy,C,C_true,res_it]=test_poisson_sor()

% Numerical method to solve
% [D_{xx}+D_{yy}]C=s(x,y),
% subject to periodic boundary conditions in the x-direction,
% and Neuman boundary conditions at y=0 and y=L_y.

aspect_ratio=2;

Ny=101;
Nx=aspect_ratio*(Ny-1)+1;

iteration_max=1000;
relax=1.5;

Ly=1.d0;
Lx=aspect_ratio*Ly;
dx=Lx/(Nx-1);
dy=Ly/(Ny-1);

kx0=2*pi/Lx;
ky0=pi/Ly;
A0=10;

dx2=dx*dx;
dy2=dy*dy;

f_source=zeros(Nx,Ny);

% Initialise source

kx=kx0;
ky=3*ky0;

for i=1:Nx
    for j=1:Ny
      x_val=(i-1)*dx;
      y_val=(j-1)*dy;
      f_source(i,j)=A0*cos(kx*x_val)*cos(ky*y_val);
```

```
40      end
41  end
42
43  % Compute analytic solution ***********************************************
44
45
46  xx=0*(1:Nx);
47  yy=0*(1:Ny);
48  C_true=zeros(Nx,Ny);
49
50  for i=1:Nx
51      for j=1:Ny
52        xx(i)=(i-1)*dx;
53        yy(j)=(j-1)*dy;
54
55        C_true(i,j)=(-A0/(kx*kx+ky*ky))*cos(kx*xx(i))*cos(ky*yy(j));
56
57      end
58  end
59
60  % Iteration step ***********************************************************
61  % Initial guess for C:
62  C=zeros(Nx,Ny);
63
64  res_it=0*(1:iteration_max);
65
66  for iteration=1:iteration_max
67
68      C_old=C;
69
70      for i=1:Nx
71
72          % Periodic BCs here.
73          if(i==1)
74              im1=Nx-1;
75          else
76              im1=i-1;
77          end
78
79          if(i==Nx)
80              ip1=2;
81          else
82              ip1=i+1;
83          end
84
```

```
85          % SOR step in here now:
86          for j=2:Ny-1
87
88              diagonal=(2.d0/dx2)+(2.d0/dy2);
89              tempval=(1.d0/dx2)*(C(ip1,j)+C(im1,j))+(1.d0/dy2)*(C(i,j+1)+C(i,j-1)
                    )-f_source(i,j);
90              C(i,j)=(1-relax)*C(i,j)+relax*tempval/diagonal;
91
92          end
93      end
94
95      % Implement Dirichlet conditions at y=0,y=L_y.
96      C(:,1)=C(:,2);
97      C(:,Ny)=C(:,Ny-1);
98
99      res_it(iteration)=max(max(abs(C-C_old)));
100
101 end
102
103 end
```

codes/poisson_matlab/test_poisson_sor.m

Lines 87-88 contain the idea of a 'sweep'. At a fixed value of i, various j-values are considered. For j=2, only 'old' values of $C(i,j)$ are known (that is, values of $C(i,j)$ at iteration level N). These old values are used to update $C(i,2)$, giving a value of $C(i,2)$ valid at iteration level $N+1$. Now, moving on to $j=3$, the new value $C^{N+1}(i,2)$ is known, along with the old value $C^N(i,4)$. These two values are used to estimate a new value of $C^{N+1}(i,3)$. In this way, the counter j 'sweeps' through all allowed j-values, and $C^N(i,j)$ is updated to a new value $C^{N+1}(i,j)$, using a combination of old and new neighbouring values in the process.

9.3 Now you try...

Exercise 9.1 *Take the Matlab code that you used previously to solve the diffusion equation. Replace the Jacobi solver with an SOR solver. Investigate the convergence of the method with respect to the parameter ω. For the optimal value of ω, compare the convergence properties with those of the Jacobi method. Be careful that your new code reproduces the old results, and again validate your Matlab code with respect to the analytical solution obtained in Chapter 4.*

Chapter 10

Introduction to Fortran

Overview

I am going to try an example-based introduction to Fortran, wherein I provide you with a sample code, and then tell you about it. I will then ask you to some tasks based on the code, and to modify it.

10.1 Preliminaries

A basic Fortran code is written in a single file with a `.f90` file extension. It conists of a **main part** together with **subroutine** definitions. A subroutine is like a subfunction in Matlab or C, with one key difference that I will explain below.

The main part

The main code is enclosed by the following declaration pair:

```
program mainprogram
...
end program mainprogram
```

At the top level, all variables that are to be used must be declared (otherwise compiler errors will ensue). Variables can be declared as integers or as double-precision numbers (other types are possible and will be discussed later on). Before variable declarations are made, a good idea is to type `implicit none`. This means that Fortran will **not** assume that symbols such as `i` have an

(implicit) type. It is best to be honest with the compiler and tell it upfront what you are going to do. Equally, it is not a good idea for the compiler to try to guess what you mean.

An array of double-precision numbers is defined as follows:

```
integer :: Nx,Ny
parameter (Nx = 201, Ny = 101)
double precision, dimension(1:Nx,1:Ny) :: my_array
```

This creates an array of double-precision numbers where the indices go from $i = 1, 2, \cdots, 201$, and $j = 1, 2, \cdots, 101$. there is nothing special in Fortran about starting arrays with $i = 1$: they can start from any integer whatsoever (positive or negative).

After defining all arrays and all other variables operations are performed on them using standard manipulations. These typically include 'do' loops (the Fortran equivalent of 'for' loops), and 'if' and 'if-else' statements. The syntax for these operatiosn is given below in the sample code (Section 10.2).

Column-major ordering

To understand column-major ordering, consider the following array:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

If stored in contiguous memory in a column-major format, this array will take the following form in memory:

$$1\ 4\ 2\ 5\ 3\ 6.$$

Suppose that elements of the array A are denoted by A_{ij} (i for rows, j for columns). When these elements are accessed sequentially in contiguous memory, it is the row index that increases the fastest. Thus, in Fortran, a `do` loop for manipulations on the array A should be written out as follows:

```
do j=1,2
  do i=1,3
    ! manipulations on A(i,j) here
    ...
  end do
end do
```

Subroutines

Subroutines contain discrete tasks that are repeated many times. Instead of having a main code that contains multiple copies of the same piece of code, such code-tasks are relegated to subroutines. The advantages are economy-of-code and computational efficiency. Unlike in C, arrays can be passed to subroutines in a blindly straightforward manner. Examples of such subroutines can be found in the sample code (Section 10.2).

As mentioned previously, a subroutine in Fortran is like a subfunction in C or Matlab. However, there is one key difference: **formally, a subroutine produces no explicit outputs**. Thus, suppose we want to operate on a variable x with an operation f to give a result y (formally, $y = f(x)$). In Fortran, we view x and y as **inputs** to a subroutine wherein y is assigned the value $f(x)$ as part of the subroutine's implementation. This will become clearer in examples.

Output

Finally, the result of these manipulations should be sent to a file, for subsequent reading. The values in an array `my_array` of size $(1, \cdots, N_x) \times (1, \cdots N_y)$ can be written to a file as follows:

```
open(unit=20,file='myfile.dat',status='UNKNOWN')

do j=1,Ny
  do i=1,Nx
    write(20,*) my_array(i,j)
  end do
  end do
close(unit=20, status='KEEP')
```

10.2 The code

The following code solves the model Poisson problem using SOR iteration. If done correctly, it should reproduce exactly the results obtained previously in Matlab. An output file called 'oned.dat' is produced. I cannot remember why I called the output file by this name. However, these things are rather arbitrary.

```
! ************************************************************

program mainprogram
        implicit none

        integer :: Nx,Ny
        parameter (Nx = 201, Ny = 101)

        double precision :: dx,dy,x_val,y_val,Lx,Ly,pi=3.1415926535
        double precision :: ax,ay,diag_val,relax,tempval,err1,err2,A0
        double precision, dimension(1:Nx,1:Ny) :: f_source,C,C_old

        integer :: i,j,im1,ip1,iteration,max_iteration=1000

        Lx=2.d0
        Ly=1.d0

        A0=10.d0

        dx=Lx/dble(Nx-1)
        dy=Ly/dble(Ny-1)

        ax=1.d0/(dx*dx)
        ay=1.d0/(dy*dy)
        diag_val=2.d0*ax+2.d0*ay
        relax=1.5d0

! ************************************************************
! compute source, initialise guess

        f_source=0.d0
        C=0.d0
        C_old=0.d0

        write(*,*) 'getting source'
        call get_f_periodic(f_source,Nx,Ny,dx,dy,Lx,Ly,A0)
        write(*,*) 'done'

```

```
39 ! ************************************************************
40 ! sor steps
41
42         do iteration=1,max_iteration
43           err1 = 0.0
44
45           ! for keeping track of the error
46           C_old=C
47
48           do j = 2,Ny-1
49             do i = 1,Nx
50
51               if(i.eq.1)then
52                 im1=Nx-1
53              else
54                 im1=i-1
55              end if
56
57               if(i.eq.Nx)then
58                 ip1=2
59               else
60                 ip1=i+1
61               end if
62
63               tempval=ax*(C(ip1,j)+C(im1,j))+ay*(C(i,j+1)+C(i,j-1))-f_source(i,j
                     )
64               C(i,j)=(1-relax)*C(i,j)+relax*tempval/diag_val
65
66             end do
67           end do
68
69           ! Implement Dirichlet conditions at y=0,y=L_y.
70           C(:,1)=C(:,2)
71           C(:,Ny)=C(:,Ny-1)
72
73           if(mod(iteration,100)==0)then
74             call get_diff(C,C_old,Nx,Ny,err1)
75             write(*,*) iteration, ' Difference is ', err1
76           end If
77
78         end do
79
80         write(*,*) ' Difference is ', err1
81
82
```

```
83 ! ************************************************************
84 ! write result to file
85
86           write(*,*) 'writing to file'
87           open(unit=20,file='oned.dat',status='UNKNOWN')
88
89           do j=1,Ny
90             do i=1,Nx
91               x_val=(i-1)*dx
92               y_val=(j-1)*dy
93               Write(20,*) x_val, y_val, C(i,j)
94             end do
95           end do
96           close(unit=20, status='KEEP')
97           write(*,*) 'done'
98
99 end program mainprogram
100
101
102 ! ************************************************************
103 ! ************************************************************
104
105   subroutine get_f_periodic(f_src,Nx,Ny,dx,dy,Lx,Ly,A0)
106   implicit none
107
108   integer ::i,j,Nx,Ny
109   double precision :: dx,dy,Lx,Ly,x_val,y_val, pi=3.1415926535
110   double precision :: kx0,ky0,kx,ky,A0
111   double precision :: f_src(1:Nx,1:Ny)
112
113   kx0=2.d0*pi/Lx
114   ky0=pi/Ly
115
116   kx=kx0
117   ky=3.d0*ky0
118
119   f_src=0.d0
120   do j=1,Ny
121     do i=1,Nx
122       x_val=(i-1)*dx
123       y_val=(j-1)*dy
124       f_src(i,j)=A0*cos(kx*x_val)*cos(ky*y_val)
125     end do
126   end do
127
```

```
128   return
129   end subroutine get_f_periodic
130
131 ! ************************************************************
132
133   subroutine get_diff(C,C_old,Nx,Ny,diff)
134   implicit none
135
136   double precision :: diff,sum
137   integer :: Nx,Ny,i,j
138   double precision, dimension(1:Nx,1:Ny) :: C, C_old
139
140   sum = 0.0D0
141   Do j = 1, Ny
142     Do i = 1, Nx
143       sum = sum + (C(i,j)-C_old(i,j))**2
144     End Do
145   End Do
146   diff = sum
147
148   Return
149   End subroutine get_diff
150
151 ! ************************************************************
152 ! ************************************************************
```

codes/poisson_code/main_periodic_sor.f90

10.3 Porting Output into Matlab

It can be useful to examine the data in a file such as 'oned.dat' in Matlab. There are many ways of doing this. Below is my favourite way:

```
function [X,Y,C]=open_dat_file()

% We need to specify the size of the computational domain, as this can't be
% inferred from the datafile.

Nx=201;
Ny=101;

% Here I create a character array called "filename". This should
% correspond to the name of the Fortran-generated file.

filename='oned.dat';

% Here is the number of lines in the datafile.
n_lines=Nx*Ny;

% Open the file. Here, fid is a label that labels which line in the file
% is being read. Obviously, upon opening the file, we are at line 1.

fid=fopen(filename);

% Preallocate some arrays for storing the data.

X=0*(1:n_lines);
Y=0*(1:n_lines);
C=0*(1:n_lines);

% Loop over all lines.

for i=1:n_lines
    % Grab the data from the current line. Once the data is grabbed, the
    % label fix automatically moves on to the next line.
    % The data from the current line is grabbed into a string - here called
    % c1.
    c1=fgetl(fid);

    % Next I have to convert the three strings on any given line into three
    % doubles. This is done by scanning the string into an array of
    % doubles, using the "sscanf" command:
    vec_temp=sscanf(c1,'%f');
```

```
41
42      % Now it is simple: just assign each double to a value x, y, or C.
43      x_temp=vec_temp(1);
44      y_temp=vec_temp(2);
45      C_temp=vec_temp(3);
46
47      % Read the x-, y-, and C-values into their own arrays.
48      X(i)=x_temp;
49      Y(i)=y_temp;
50      C(i)=C_temp;
51  end
52
53  % Finally, reshape these arrays into physical, two-dimensional arrays.
54
55  X=reshape(X,Nx,Ny);
56  Y=reshape(Y,Nx,Ny);
57  C=reshape(C,Nx,Ny);
58
59  % Important! Close the file so that it is not left dangling.  Not closing a
60  % file properly means that in future, it will be difficult to manipulate
61  % it.  For example, it is impossible to delete or rename a a currently-open
62  % file.
63
64  fclose(fid);
65
66  end
```

codes/poisson_code/open_dat_file.m

This file should be stored int he same directory as 'oned.dat'. Then, at the command line, type

```
[X,Y,C]=open_dat_file();
```

The results can be visualized as usual:

```
[h,c]=contourf(X,Y,C);
set(c,'edgecolor','none')
```

Provided the source function and domain size are the same in both cases, this figure should agree exactly with the one generated previously using only Matlab (Figure 10.1). Here,

$$s(x, y) = A_0 \cos(k_x x) \cos(k_y y), \tag{10.1}$$

with $k_x = k_{x0}$ and $k_y = 3k_{y0}$, and $A_0 = 10$. Further details: $k_{x0} = (2\pi/L_x)$ is the fundamental wavenumber in the x-direction, and $k_{y0} = \pi/L_y$ is the fundamental wewavenumber in the y-direction. The domain geometry is chosen to be $L_x = 2$ and $L_y = 1$.

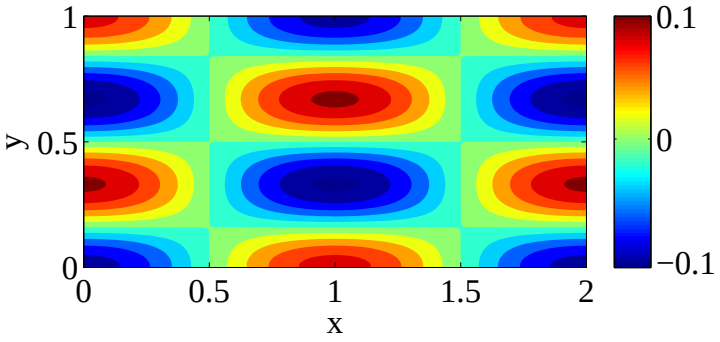


Figure 10.1: Solution of the Poisson problem for the source (10.1). Grid size: $N_x = 201$ and $N_y = 101$.

10.4 Now you try...

The next assignment involves running a job on a cluster, rather than your local PC. First, your `.f90` code must be turned into a machine-readable executable. This is done using a **compiler**. Here is a sample compile line:

```
ifort main_periodic_sor.f90 -o sor.x
```

Here, the text file `main_periodic.f90` is being turned into an exectuable `sor.x` using the 'ifort' compiler.

When you have a compiled executable, **DO NOT** execute on the command line. This is an extremely ignorant form of behaviour. Instead, to ensure faireness to fellow users, you should use the cluster's queuing system. This is a job scheduler that ensures that jobs are executed through a queue, in a afair manner. Thus, the next step instead is to write a **submit scrpit** containing information that the queue scheduler can use to submit your job. Here is a sample submit script for a particular queue on the Maths mini-hpc cluster Orr:

```
#!/bin/bash
#$ -S /bin/bash
#$ -N job_sor
#$ -q 4x8a.q
#$ -pe orte 8
#$ -V
#$ -cwd

cd /home/lennon/ucd_summerschool/main_periodic_sor

./sor.x
```

I called this file 'submit_lon' (no extension).

Common Programming Error:

Writing a submit script in DOS (Windows) and expecting it to run in a Linux environment. Both systems use different formats for spaces and returns. Thus, a script written under DOS should be run through the linux command 'dos2unix' before submission to queue.

Then, to send your job off to the queue, type the following string at the command line:

```
qsub submit_lon
```

You can check on the status of your job by typing

```
qstat
```

Now try the following exercise.

Exercise 10.1 *Run the Fortran code for the Poisson problem on the cluster. You will need a* **submit script**, *which you then submit to a queue. The result will be stored in a file called 'oned.dat', which you should analyse in Matlab. Finally, diagnostic data realted to the code execution (standard output) will be put into a separate file labelled by the job identifier.*

Chapter 11

Challenge problem in Fortran

Overview

In this chapter we shall write a Fortran code to solve the model diffusion equation. We shall output data sequentially to text files (not to be overwritten). We shall then postprocess the data, reading raw data into Matlab to make contour plots, movies, and other diagnostics.

11.1 The Fortran task

Exercise 11.1 *Write a Fortran code to solve the model diffusion equation with a source. Data should be periodically dumped to text files. When complete, postprocess the data as in the next section.*

Dumping data periodically to a file

In order to dump data periodically to a file, a variable of 'character' type needs to be defined. This is done along with the definition of the other variables, at the beginning of the main part of the code:

```
character*60 filename
```

Thus, a variable of 'character' type is created, with the name 'filename'. A character datatype can be thought of as an array of strings. Thus, `character*60 filename` is a string that can hold up to 60 individual characters.

Suppose now that we are in an iterative loop, where the iteration is over time steps, labelled here by the counter `iteration_time`. Suppose further that every `file_period_time` timesteps, we wish to output a certain field variable C (together with relevant coordintes (x, y)) to a datafile, such that no datafile is overwritten. Here is a way to do it:

```
! ************************************************
          ! write result to file

          if(mod(iteration_time,file_period_time).eq.0)then
            write(*,*) 'writing to file'
            write(filename,'(A,I9,A)')'diffusion_',iteration_time,'.dat'
            write(*,*) filename
            open(unit=9,file=filename,status='unknown')

            do j=1,Ny
              do i=1,Nx
                x_val=(i-1)*dx
                y_val=(j-1)*dy
                write(9,*) x_val, y_val, C(i,j)
              end do
            end do

            close(9)
            write(*,*) 'done'
          end if

! ************************************************
```

11.2 Making movies with Matlab

The first Matlab code here takes all of the datafiles generated by the Fortran code and turns them into lovely Matlab figures. This is done in the background, and the figures are saved for future use. The Matlab function operates sequentially on each `.dat` file generated, assuming all such files are in the current working directory:

```
function []=create_all_fig_files()

% Declar these to be global variables so they don't need to be passed to
% subfunctions.

global Nx
global Ny
global n_lines

% ************************************************************************
% File structure information

Nx=201;
Ny=101;

n_lines=Nx*Ny;

% ************************************************************************
% Grab all files in folder
% Here, files with names of the form "diffusion_*.dat" are examined.  The
% filenames of all such files are grabbed into a cell array called "files".
% The number of entries in the cell array is grabbed using the "numel"
% command.

files=dir('diffusion_*.dat');
klim=numel(files);

% ************************************************************************
% Initialize Matlab figure for plotting

% Here, I open a figure (h is a handle to a figure).
% However, I make it invisible - it is only defined in the background.
% This is a good idea for large jobs, especially if such jobs are to be run
% on a cluster with matlab.

h=figure;
set(h, 'Visible','off');

```

```
% Loop over all files from kk=1 up to the last file with index klim.

for kk=1:klim

    % Count the file and output it to the screen as a string message.
    display(strcat('drawing ',num2str(kk),'th file'))

    % Grab the name of the kkth file from the cell array.
    filename=files(kk).name;

    % Load the data from the kkth file into arrays of double-precision
    % numbers.

    % Here, the subfunction my_openfile() is very similar to the one we had
    % before: it opens the file of name "filename" and reads sequentially
    % through the lines of the file, gathering the data into vectors.  The
    % vectors are reshaped into two-dimensional arrays at the end of the
    % operation.

    [X,Y,C]=my_openfile(filename);

    % Standard contour plotting now.
    [~,cc]=contourf(X,Y,C);
    set(cc,'edgecolor','none')
    colorbar
    set(gca,'fontname','times new roman','fontsize',18)
    xlabel('x')
    ylabel('y')

    % Create a file name for the kkth figure.
    figfilename=strcat('contour_',num2str(kk),'.fig');

    % Save the kkth contour plot to a Matlab ".fig" file.
    saveas(h,figfilename)
    clf
end

end

% ************************************************************************
% ************************************************************************

function [X,Y,C]=my_openfile(filename)

% Global variables - I did not need to pass them explilcitly to the
```

```
84 % subfunction.
85
86 global Nx
87 global Ny
88 global n_lines
89
90 fid=fopen(filename);
91
92 X=0*(1:n_lines);
93 Y=0*(1:n_lines);
94 C=0*(1:n_lines);
95
96 for i=1:n_lines
97     c1=fgetl(fid);
98     vec_temp=sscanf(c1,'%f');
99     x_temp=vec_temp(1);
100    y_temp=vec_temp(2);
101    C_temp=vec_temp(3);
102
103    X(i)=x_temp;
104    Y(i)=y_temp;
105    C(i)=C_temp;
106 end
107
108 X=reshape(X,Nx,Ny);
109 Y=reshape(Y,Nx,Ny);
110 C=reshape(C,Nx,Ny);
111
112 fclose(fid);
113
114 end
115
116 % ************************************************************************
```

codes/diffusion_code/create_all_fig_files.m

These are invisible figure files! The can be opened as follows:

```
openfig('contour_1.fig','new','visible')
```

Also, they can be made into a movie:

```
function []=make_move1()

%**************************************************************************

screen_size = get(0, 'ScreenSize');

klim=40;
numframes = klim;

%**************************************************************************
%% Movie code

% Here, I open a figure and set the size of this figure.
fig1 = figure(1);

x1=0*screen_size(3);
y1=0.1*screen_size(3);
x2=0.6*screen_size(3);
y2=0.7*screen_size(4);

set(fig1, 'Position', [x1 y1 x2 y2 ] );

% Next, I create a Matlab array called A_mov.  This array will store a
% separate contour plot.  These contourplots make up the frame of the
% movie.

% The first index of the array counts the contour plot (or the frame).
% The second index stores the actual figure or contour plot.

A_mov = moviein(numframes, fig1);
set(fig1, 'NextPlot', 'replacechildren')

%**************************************************************************

for k=1:klim

    time=0.1*k;

    filename1=strcat('contour_',num2str(k),'.fig');

    % Here, I open the kth contourplot into a temporary figure called
    % "tempfig".

    tempfig=openfig(filename1,'reuse','visible');
```

```
    set(tempfig, 'Position', [x1 y1 x2 y2 ]);
    title(strcat('t=',num2str(time)))

    % Now I place the temporary figure "tempfig" into the kth frame of the
    % movie array.

    A_mov(:,k) = getframe(tempfig);
    close(tempfig)

    display(strcat('k=',num2str(k)))
end

% Here is the Matlab command to turn the movie array into a replayable
% movie.

movie(fig1, A_mov,1,3);
save movie.mat A_mov;

% Finally, I turn the Matlab movie into a standard portable avi file.
% This part can be very tricky, and to get it to work across platforms,
% sometimes it is necessary to specify a certain codec.  Trial and error
% and googling are required here.



%%  Example with a codec specified, and with compression.:
% movie2avi(A_mov,'movie_compressed.avi','compression','Cinepak','quality',100,'
    fps',1);

%% Working example (on my computer):

movie2avi(A_mov, 'movie.avi', 'compression', 'None','fps',1);

%% In both examples, "fps" is a good parameter to adjust.  It is the
%% number of movie frames that gets played per second.

% ********************************************************************

end
```

codes/diffusion_code/make_move1.m

11.3 Further diagonsitics

Exercise 11.2 *Let $C(x, y, t)$ denote the solution to the model diffusion equation with suitable initial data, and let $C_0(x, y)$ be the solution of the model Poisson problem. We have already shown analytically that*

$$\lim_{t\to\infty} C(x, y, t) = C_0(x, y). \tag{11.1}$$

In addition to this analytical result, modify the Matlab postprocessing routines in this chapter so that they can be used to confirm Equation (11.1).

Hint: With very little effort, and very little modification of `create_all_fig_files.m`, I obtained the following figure (Figure 11.1):

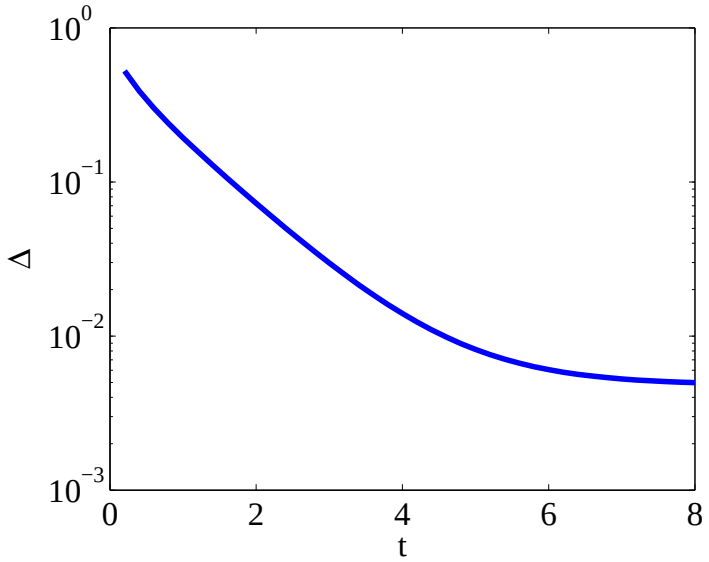


Figure 11.1: Decay of the L^2 norm of the deviation away from the stationary solution.

Here, on the y-axis, I have plotted

$$\Delta(t) = \left\{ \frac{1}{N_x N_y} \sum_{i,j} \left[C(i, j, t) - C_0(i, j) \right]^2 \right\}^{1/2},$$

where I obtained $C_0(x, y)$ analytically. In fact, the decay of $\Delta(t)$ in Figure 11.1 should be strictly exponential. Perhaps with a finer grid resolution or a smaller timestep, this can be demonstrated.

Chapter 12

Introduction to shared memory

Overview

Recall from Section 3 our discussion about modern desktop computers: although such computers still have a single CPU, they possess two (or more) processing units (or cores), which are placed on the same chip. The cores share some cache ('L2 cache'), while some other cache is private to each core ('L1 cache'). This enables the computer to break up a computational task into two(or more) parts, work on each task separately, via the private cache, and communicate necessary shared data via the shared cache. This architecture therefore facilitates **parallel computing**, thereby speeding up computation times. High-level programs such as MATLAB take advantage of multiple-core computing without any direction from the user. On the other hand, lower-level programming languages such as Fortran require explicit direction from the user in order to implement multiple-core processing. The aim of this chapter is to do precisely this, usingthe OpenMP standard.

Again, for clarity, we repeat some things: we reserve the word **processor** for the entire chip, which will consist of multiple sub-units called **cores**. Sometimes the cores are referred to as **threads** and this kind of computing is called **multi-threaded**.

12.1 Shared memory – concepts

A useful schematic diagram of shared memory is the following one, obtained on Wikipedia (Figure 12.1):

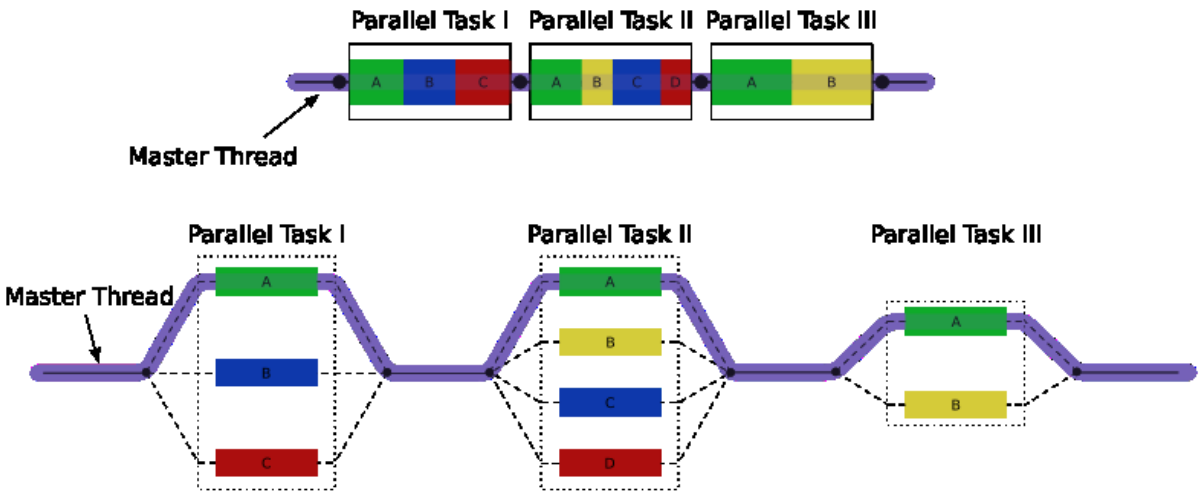


Figure 12.1: Schematic diagram of a multithreaded calculation

The idea is that a code in the absence of multithreading trivially consists of a single thread called the **master**. Then, when a particularly heavy piece of computation needs to be done (such as a large nested 'do' loop), the master breaks the task into many threads. Different parts of the calculation are done on different threads. If the calculation is non-local (also, if the amount of data operated on is small), such that on thread X, data from thread Y is required, then all such data is stored in the L2 cache, to be shared between X and Y and all other threads. On the other hand, data which is strictly local to thread X can be stored in the L1 cache of X. Finally, when all the threads have done their part of the calculation, they rejoin the master. At this end-stage, the master must have access to the necessary data in the L2 cache.

The advantage of such a concept is the existence of a set of simple directives (understandable to a Fortran compiler) to implement multi-threading. The purpose of this chapter is to explain these directives to you. However, a disadvantage of this approach is the limitation imposed by the computer architecture: we can only access as much parallelism as there are cores on the computer. A desktop will have 2 or 4 cores; some machines will have 8 or even 16 cores. Thus, a theoretical speedup of 16 is possible, but no higher. Further limitations:

- The finite bandwidth of the bus that enables communication between the memory of the individual threads (speedup limitation)

- The finite amount of global memory accessible by all the threads (jobsize limitation)

These severe architecture-dependent limitations are overcome by moving to a more sophisticalted parallelism called MPI, beyond the scope of this course.

12.2 OMP directives

To split up a Fortran 'do loop' into threads, it suffices to put a special line before and after the loop, and to decide which variables are thread-private and which variables are shared. Typically,

- Arrays that are operated on are shared.
- Intermediate variables and loop variables (`i`, `j` etc) are private.

Common Programming Error:

Not specifiying which variables are private and which are shared – can lead to catastrophic errors.
Not initializing private variables. At thread-creation time, all private variables have no value and must be assigned some initial value explicitly in the code.

An example of the relevant OMP directives is shown here. This piece of code assigns the array `f_src` some value.

```
!$omp parallel default(shared), private(i,j,x_val,y_val)
!$omp do
  do j=1,Ny
    do i=1,Nx
      x_val=i*dx-(dx/2.d0)
      y_val=j*dy-(dy/2.d0)
      f_src(i,j)=-A0*cos(kx*x_val)*cos(ky*y_val)
    end do
  end do
!$omp end do
!$omp end parallel
```

The convention here is that all variables are shared – unless indication is given to the contrary. Thus, the private variables are only those intermediate variables that are useed in the construction of the final answer. These private variables are `x_val`, `y_yval`, `i`, and `j`.

12.3 OMP and SOR

It would appear that to do SOR with OMP, it suffices blindly to stick a few lines of code in front of the relevant 'do' loops. However, this could be dangerous, and lead to the failure of the SOR

algorithm. Suppose that an array C_{ij} is being obtained by SOR iteration, such that some old value C_{ij}^{old} is being replaced by a new improved value C_{ij}^{new}. Assume that for a fixed `j`, a sweep is performed, starting at `i=1`. Thus, symbolically,

$$C_{ij}^{\text{new}} = f\left(C_{i-1j}^{\text{new}}, C_{i+1j}^{\text{old}}\right),$$

where f is some linear function that depends on the particular matrix problem being solved. Now suppose that this job has been split up into threads that share the array C via shared memory. Suppose that C_{ij} is being operated on using thread X. For some `i`, it may be the case that C_{i-1j}^{new} is operated on using thread $Y \neq X$. Thread X may 'race ahead' of thread Y, such that C is replaced by C^{new} slowly on thread Y compared to thread X. Thus, thread Y may use C_{i-1j}^{old} to update C_{ij}, instead of C_{i-1j}^{new}, possibly causing the failure of the SOR method and non-convergence. This is an example of a so-called 'race condition' on OMP, and is to be avoided if possible (they can be catastrophic).

Common Programming Error:

Not noticing a race condition – they are common and not restricted to the SOR algorithm. They typically involve arrays whose elements are being updated with other elements from the same array.

Red-black coloring

The race condition for SOR can be overcome by implementing something called red-black coloring. This works only for problems (such as the diffusion problem) where the discretization uses only nearest neighbours (however, if neighbours other than the nearest neighbours are needed, then a scheme with more colours than just red and black will work). The SOR sweep is split into two half-steps. During the first half-step, and for fixed `j`, grid sites with even `i` are updated. During the next half-step, grid sites with odd `i` are updated.

Alternatively, for fixed `j`, grid sites with even `i` can be thought of as being 'red', while grid sites with odd `i` can be thought of as being 'black'. During the first half-sweep, red sites are updated with the old black values (consistent with the SOR algorithm), and during the second half-seep, the black sites are updated with the just-updated (new) red values.

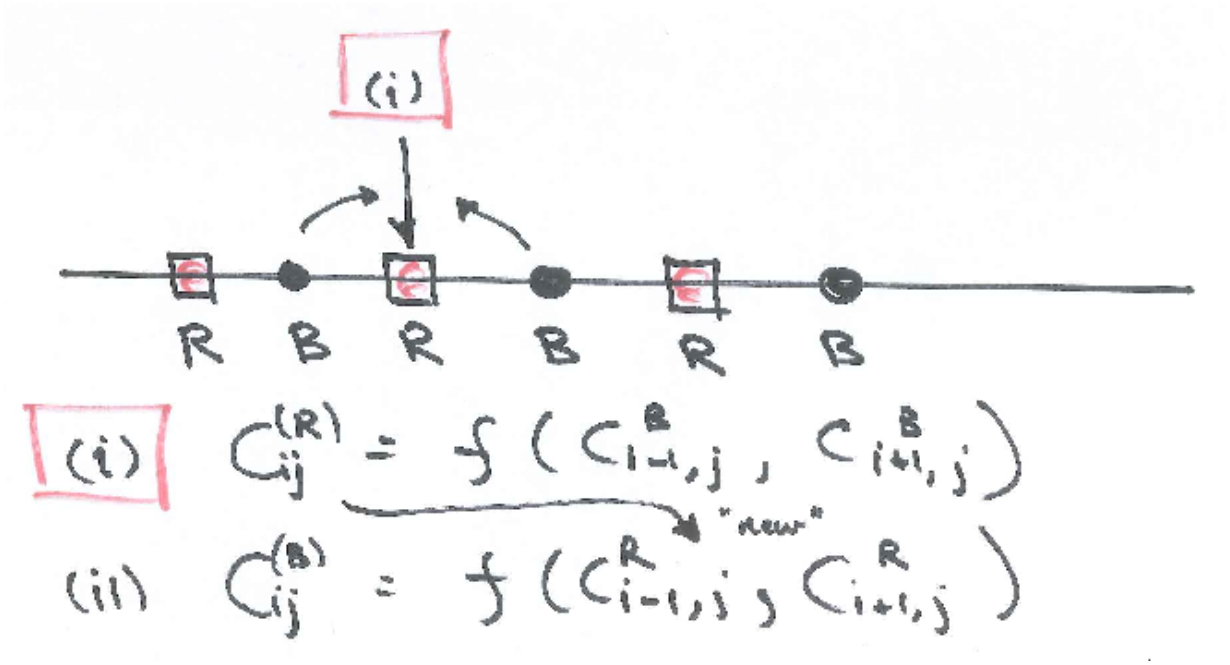


Figure 12.2: Schematic diagram of red-black coloring for the SOR algorithm

The advantage of this approach is synchronization: the second half-sweep does not begin until the first half-sweep has been completed by all threads. The first half-sweep is straightforward: red values are updated with old black values. This sweep is implemented on all threads which then join the master. Then, during the second half-sweep, the threads fork again and the black values are updated with a consistent set of (new) red values.

12.4 Other simple OMP tricks

You may have noticed by now that Fortran (in its '.f90' incarnation) supports vectorization. That is, for a square array C with

$$i = 1, 2, \cdots, n_x, \qquad j = 1, 2, \cdots, n_y, \tag{12.1}$$

the operation

```
do j=1,n_y
  do i=1,n_x
    C(i,j)=1.d0
  end do
end do
```

can be implemented equivalently as

```
  C=1.d0
```

This is still an array operation, whose speed of implemnetation can be increased with OpenMP. The relevant directive is called a **workshare**:

```
!$omp parallel workshare
  C=1.d0
!$omp end parallel workshare
```

Finally, there are some situations where an array needs to be populated using a very simple formula, with no need for intermediate or temporary variables. Suppose that we wanted to create an array

$$D_{ij} = \frac{1}{\Delta x^2}\left(C_{i+1,j} + C_{i-1,j} - 2C_{ij}\right) + \frac{1}{\Delta z^2}\left(C_{i,j+1} + C_{i,j-1} - 2C_{ij}\right) \tag{12.2}$$

to store a numerical approximation of the Laplacian of C, where C is a square array with the same indices as in Equation (12.1). Of course, it is not possible to compute the Laplacian at bounary points, so we do not even attempt such a thing. The operation in Equation (12.2) can be implemented using a parallel `forall` loop as follows:

```
!$omp parallel workshare
  forall (i = 2:nx-1, j=2:ny-1)
    D(i,j)=(1.d0/dx*dx)*(C(i+1,j)+C(i-1,j)-2.d0*C(i,j)) &
     +(1.d0/dy*dy)*(C(i,j-1)+C(i,j+1)-2.d0*C(i,j))
  end forall
!$omp end parallel workshare
```

Note the use of a **continuation character** to enable a line to be broken, for ease of reading.

12.5 OMP reduction

Sometimes an operation is performed on each thread, with a certain local result, and it is necessary to combine all such local results back into the master thread. Examples of this kind include sums and maxima. For example, suppose that an array C is split up between threads, such that thread i operates only on a chunk of the array, say $C^{(i)}$. Then, suppose that on thread i, the following sum is computed

$$s_i = \sum_{pq} C^{(i)}_{pq}.$$

It might become necessary during the course of an assignment to compute

$$s = \sum_i s_i, \qquad \text{sum over all threads.}$$

This is an example of an OMP **reduction**. The syntax for this operation is given in the following example.

```
  subroutine get_diff(u,u_old,maxl,maxn,diff)
  implicit none

  double precision :: diff,sum
  integer :: maxl,maxn,i,j,tid
  double precision, dimension(0:maxl,0:maxn) :: u, u_old

  sum = 0.0D0
!$omp parallel do default(none), &
!$omp   private(i,j), shared(u,u_old,maxn,maxl,sum), &
!$omp   reduction(+:sum)

  Do j = 1, maxn-1
    Do i = 1, maxl-1
      sum = sum + (u(i,j)-u_old(i,j))**2
    End Do
  End Do
!$omp end parallel do
  diff = sum

  Return
  End subroutine get_diff
```

Common Programming Error:

Reduction can only be done on shared variables.

Chapter 13

Multithreading for the model Poisson equation

Overview

We solve the model Poisson problem numerically using OpenMP. We test the code's parallel efficiency. You will then be asked to implement OMP parallelism for the model diffusion problem.

13.1 The code for the model Poisson problem

```
1  ! ************************************************************
2
3  program mainprogram
4          implicit none
5
6          integer :: Nx,Ny
7          parameter (Nx = 201, Ny = 101)
8
9          double precision :: dx,dy,x_val,y_val,Lx,Ly,pi=3.1415926535
10         double precision :: ax,ay,diag_val,relax,tempval,err1,err2,A0
11         double precision, dimension(1:Nx,1:Ny) :: f_source,C,C_old
12
13         integer :: i,j,im1,ip1,iteration,max_iteration=1000
14
15         Lx=2.d0
16         Ly=1.d0
17
18         A0=10.d0
19
```

```
20          dx=Lx/dble(Nx-1)
21          dy=Ly/dble(Ny-1)
22
23  ! ************************************************************
24  ! compute source, initialise guess
25
26          f_source=0.d0
27          C=0.d0
28          C_old=0.d0
29
30          write (*,*) 'getting source'
31          call get_f_periodic(f_source,Nx,Ny,dx,dy,Lx,Ly,A0)
32          write (*,*) 'done'
33
34  ! ************************************************************
35  ! sor steps
36
37          do iteration=1,max_iteration
38
39            ! for keeping track of the error
40  !$omp parallel workshare
41              C_old=C
42  !$omp end parallel workshare
43
44              ! First sweep
45              call do_sor_C(C,f_source,dx,dy,Nx,Ny,0)
46              ! Second sweep
47              call do_sor_C(C,f_source,dx,dy,Nx,Ny,1)
48
49              ! Implement Dirichlet conditions at y=0,y=L_y.
50              C(:,1)=C(:,2)
51              C(:,Ny)=C(:,Ny-1)
52
53            if(mod(iteration,100)==0)then
54              call get_diff(C,C_old,Nx,Ny,err1)
55              write(*,*) iteration, ' Difference is ', err1
56            end If
57
58          end do
59
60          write(*,*) ' Difference is ', err1
61
62
63  ! ************************************************************
64  ! write result to file
```

```
65
66            write(*,*) 'writing to file'
67            open(unit=20,file='poisson.dat',status='UNKNOWN')
68
69            do j=1,Ny
70              do i=1,Nx
71                x_val=(i-1)*dx
72                y_val=(j-1)*dy
73                Write(20,*) x_val, y_val, C(i,j)
74              end do
75            end do
76            close(unit=20, status='KEEP')
77            write(*,*) 'done'
78
79  end program mainprogram
80
81
82  ! ************************************************************
83  ! ************************************************************
84
85    subroutine get_f_periodic(f_src,Nx,Ny,dx,dy,Lx,Ly,A0)
86    implicit none
87
88    integer ::i,j,Nx,Ny
89    double precision :: dx,dy,Lx,Ly,x_val,y_val, pi=3.1415926535
90    double precision :: kx0,ky0,kx,ky,A0
91    double precision :: f_src(1:Nx,1:Ny)
92
93    kx0=2.d0*pi/Lx
94    ky0=pi/Ly
95
96    kx=kx0
97    ky=3.d0*ky0
98
99    f_src=0.d0
100   do j=1,Ny
101     do i=1,Nx
102       x_val=(i-1)*dx
103       y_val=(j-1)*dy
104       f_src(i,j)=A0*cos(kx*x_val)*cos(ky*y_val)
105     end do
106   end do
107
108   return
109   end subroutine get_f_periodic
```

```
110
111 ! ************************************************************
112
113   subroutine get_diff(C,C_old,Nx,Ny,diff)
114   implicit none
115
116   double precision :: diff,sum
117   integer :: Nx,Ny,i,j
118   double precision, dimension(1:Nx,1:Ny) :: C, C_old
119
120   sum = 0.0D0
121 !$omp parallel do default(none), &
122 !$omp    private(i,j), shared(C,C_old,Ny,Nx), &
123 !$omp    reduction(+:sum)
124   Do j = 1, Ny
125     Do i = 1, Nx
126       sum = sum + (C(i,j)-C_old(i,j))**2
127     End Do
128   End Do
129 !$omp end parallel do
130   diff = sum
131
132   Return
133   End subroutine get_diff
134
135 ! ************************************************************
136 ! ************************************************************
137 ! ************************************************************
138
139         subroutine do_sor_C(C,f_source,dx,dy,Nx,Ny,flag)
140         implicit none
141
142         integer ::i,j,ip1,im1,Nx,Ny,flag
143         double precision :: dx,dy,ax,ay
144         double precision :: f_source(1:Nx,1:Ny),C(1:Nx,1:Ny)
145
146         double precision :: relax,diag_val,tempval
147
148         ax=1.d0/(dx*dx)
149         ay=1.d0/(dy*dy)
150         diag_val=2.d0*ax+2.d0*ay
151         relax=1.5d0
152
153 !$omp parallel default(shared), private(i,j,im1,ip1,tempval)
154 !$omp do
```

```
155         do  j=2,Ny-1
156           do  i=mod(j+flag,2)+1,Nx,2
157
158             if(i.eq.1)then
159               im1=Nx-1
160             else
161               im1=i-1
162             end if
163
164              if(i.eq.Nx)then
165               ip1=2
166              else
167               ip1=i+1
168              end if
169
170              tempval=ax*(C(ip1,j)+C(im1,j))+ay*(C(i,j+1)+C(i,j-1))-f_source(i,j
                   )
171              C(i,j)=(1-relax)*C(i,j)+relax*tempval/diag_val
172
173           end do
174         end do
175 !$omp end do
176 !$omp end parallel
177
178         end subroutine do_sor_C
```

codes/poisson_code_omp/poisson_sor.f90

13.2 Execution

On Orr, the compile line command is the following:

```
ifort -openmp poisson_sor1.f90 -static -o poisson_omp.x
```

However, this will differ from compiler to compiler. A modified submit script is required. The submit script will set an **environment variable** that specifies the number of OMP threads. Obviously, this should never exceed the number of actual threads on the computer. A sample submit script on Orr is the following:

```
#!/bin/bash
#$ -S /bin/bash
#$ -N poiss_omp
#$ -q 4x8a.q
#$ -pe orte 8
#$ -V
#$ -cwd

export OMP_NUM_THREADS=8

cd /home/lennon/ucd_summerschool/poisson_code_omp

./poisson_omp.x
```

which submits to a machine with four processors, each of which has 8 threads. Submission is then via the ordinary queue.

13.3 OMP reduction – revsited

Recall in Chapter 12 we examined a method to reduce a sum over all threads. It is instructive to consider an alternative method for doing the OMP reduction in the subroutine get_diff:

```
        subroutine get_diff(C,C_old,Nx,Ny,diff_val)
        implicit none

        integer :: Nx,Ny,i,j
        double precision, dimension(1:Nx,1:Ny) :: C, C_old

        integer :: large,tid,numthreads,OMP_GET_THREAD_NUM,OMP_GET_NUM_THREADS
        parameter (large=100)
        double precision :: diff_vec(0:large),diff_val

!$omp parallel
        numthreads = OMP_GET_NUM_THREADS()
!$omp end parallel

!$omp parallel default(shared), private(i,j,tid,diff_val)
        tid=OMP_GET_THREAD_NUM()
        diff_val=0.d0
!$omp do
        Do j = 1, Ny
          Do i = 1, Nx
            diff_val = diff_val + (C(i,j)-C_old(i,j))**2
          End Do
        End Do
!$omp end do
        diff_vec(tid)=diff_val
        write(*,*) 'tid= ',tid,'diff= ',diff_val, 'num threads= ',numthreads
!$omp end parallel

        diff_val=0.d0
        do tid=0,numthreads-1
          if(diff_vec(tid).gt.diff_val)diff_val=diff_vec(tid)
        end do
```

```
Return
End subroutine get_diff
```

This subroutine makes use of some new OMP directives – and `OMP_GET_NUM_THREADS`, and `OMP_GET_THREAD_NUM`:

- When called in a parallel region,`OMP_GET_NUM_THREADS` returns an integer N, the number of threads available. It must be called in a parallel region – otherwise the answer returned will be 1.
- When called in a parallel region, `OMP_GET_THREAD_NUM` returns an integer i that labels the current thread, with $i = 0, \cdots, N-1$.

Exercise 13.1 *Implement both kinds of reduction for the model Poisson problem and compare the residuals. Given that the code is run twice, with both executions being independent, are we justified in thinking the residuals should be the same (to machine precision) in each case?*

Exercise 13.2 *Write a Fortran code from scratch that computes*

$$p(n) = \sum_{n=1}^{N} \frac{1}{n^2}.$$

Parallelize the code using OpenMP, in particular an OMP reduction for the summation.

Now, suppose however that the maximum over all rows and columns of the residual array is required. Here, a similar operation can be performed to obtain the maximum over all threads – either by explicit computation, or by OMP reduction. Notionally, the reduction takes place as follows. Suppose we have an array C, split up between threads, such that thread i operates only on a chunk of the array, say $C^{(i)}$. Then, suppose that on thread i, the following maximum is computed

$$m_i = \max_{pq} |C^{(i)}_{pq}|.$$

Suppose now we are interested in computing

$$m = \max(m_1, m_2, \cdots), \qquad \text{maximum over all threads.}$$

The syntax for this is as follows:

```
  subroutine get_diff(u,u_old,maxl,maxn,diff)
  implicit none

  double precision :: diff,max_val,temp_val
  integer :: maxl,maxn,i,j
  double precision, dimension(0:maxl,0:maxn) :: u, u_old

  max_val=0.d0
!$omp parallel do default(none), &
!$omp   private(i,j,temp_val), shared(u,u_old,maxn,maxl,max_val), &
!$omp   reduction(max:max_val)

  Do j = 1, maxn-1
    Do i = 1, maxl-1
      temp_val=abs(u(i,j)-u_old(i,j))
      if( temp_val .gt. max_val) then
        max_val=temp_val
      end if
    End Do
  End Do
!$omp end parallel do
  diff = sum

  Return
  End subroutine get_diff
```

Common Programming Error:

Using an OMP reduction on the 'max' operator in the C language. It is only defined in Fortran – all the more reason to use lovely Fortran!

13.4 Tasks – timing

OpenMP provides built-in functions to time the execution of a parallel code. In the main code, when variables are declared, one declares three further variables:

Number of threads	Time in seconds
8	0.1299
4	0.1184
2	0.1926
1	0.2251

Table 13.1: Execution times for the SOR iteration (model Poisson problem)

```
real(8) :: start_time, end_time, OMP_get_wtime
```

Consider now a given parallel task that is to be performed. Before execution, one measures the wall time:

```
start_time=OMP_get_wtime()
```

Then, the parallel segment of code is run and the wall time is measured again:

```
end_time=OMP_get_wtime()
```

The total execution time is the difference of these two snapshots:

```
write(*,*) ' Walltime is ', end_time-start_time
```

I timed the execution of the SOR code on a 201×201 grid, with 1,000 SOR iterations and tabulated the results (Table 13.1). I also plotted the same information in Figure 13.1. The results are okay

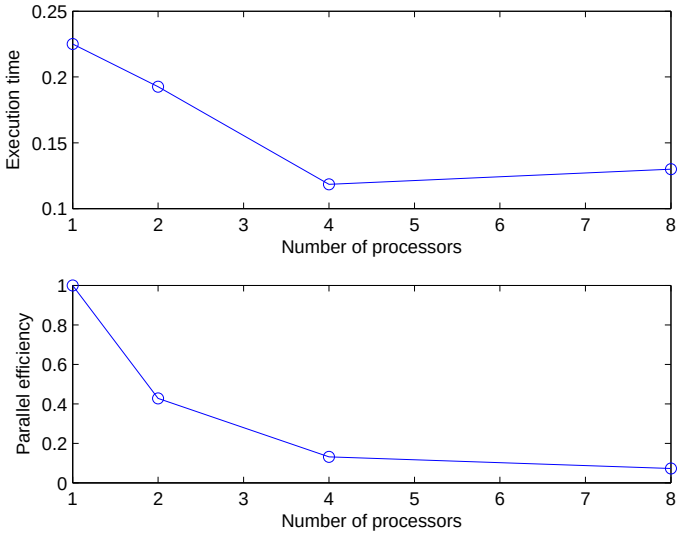


Figure 13.1: Execution times for the SOR iteration (model Poisson problem)

but not brilliant:

- For $N = 1, 2, 4$ the execution time decreases with the increase in thread count. This is good! Our code is running faster because of the multi-threading.

- However, between $N = 4$ and $N = 8$, the code is slowing down again. Thus, $N = 4$ is the optimal number of threads. going beyond $N = 4$ is a waste of time and electricity. The reasons for this can be manifold. A possibility is the existence of **communication overheads** – threads need to share data, not all of which may be in the cache. When threads spend time sharing data, it is wasted time, in other words, time that should be better spent doing the actual calculation. We say that oure code **scales** out to 4 threads, and we speak of our code's **scalability**.

- The parallel efficiency is defined to be

$$E_N = \frac{T_N}{NT_1},$$

  where N is the number of threads, T_N is the time required to run on N threads and T_1 is the time required to run on a single thread. At $N = 4$, the parallel efficiency is only 13%.

Exercise 13.3 *See if the performance of the parallel code can be improved by removing the diagnostic step involving computation of the residuals via OMP reduction. This is a purely diagnostic step and is not needed for the successful running of the code.*

13.5 Over to you...

Exercise 13.4 *Recall in Chapter 11, you wrote a Fortran code to solve the model diffusion equation with a source. Now parallelize the code in OMP. If possible, measure its performance and scalability.*

Chapter 14

Memory allocation in Fortran

Overview

We discuss dynamic memory allocation, particularly suitable for large arrays. We extend the model Poisson problem to three dimensions and solve it numerically. We develop postprocessing tools for handling the large files produced from such three-dimensional calculations.

14.1 Dynamic versus static; heap versus stack

So far in Fortran we have declared variables to be of a definite type, for example, integers, doubles, and characters. Additionally, we have created arrays of doubles, where these arrays are of a **fixed size**. The syntax for the array allocation is really quite definite and hints at the immutability of arrays so created:

```
integer :: Nx,Ny
parameter (Nx = 201, Ny = 101)
double precision, dimension(1:Nx,1:Ny) :: C
```

Here, the array C gets a fixed size that cannot be altered. We say that the array C is **static**. Because the amount of memory necessary for the creation of this array is known precisely, the array C can be placed in that highly-structured part of memory known as the **stack** (See Chapter 3). In contrast, there are situations where static arrays are not desirable:

- If the size of the array C needs to change in the course of a calculation;
- If the size of the array C is simply too large to fit in the stack.

In these situations, it is better to allocate the array C **dynamically**, on the **heap**. The syntax for the dynamic allocation of the array C is shown here:

```
        integer :: Nx,Ny
        parameter (Nx = 201, Ny = 201, Nz=101)
        double precision, allocatable, dimension(:,:) :: C

        ...
        ...

! ************************************************************
! Allocate variables

        allocate(C(1:Nx,1:Ny))

        C=0.d0

! ************************************************************
```

When the array C is no longer needed it needs to be **deallocated**, thereby freeing up space in memory:

```
        deallocate(C)
```

Typically, this is done at the end of the main code, but this is not necessarily the case.

Performance issues

Memory allocation and deallocation is expensive. Allocation on the heap is typically avoided. In particular, dynamic memory allocation in subroutines is a bad idea, because such subroutines tend to be called repeatedly. In a worst-case scenario, one would have a dynamically-allocated array in a subroutine that is called many times, thereby creating successive calls to allocate and free up memory, over and over again. For this reason, dynamic memory allocation tends only to be used in the main part of the code, and the allocation is done on a once-off basis.

Good Programming Practice:

When dealing with large arrays that are passed to subroutines, my practice is to allocate the array dynamically in the main code, and then to allocate copies of the array statically in any subroutines. Then, if I get a segmentation fault due to an excessively large request for memory that does not exist, I can copy the chunk of code for dynamic array allocation into the subroutine.

Of course, having dynamic array allocation in a subroutine that is called repeatedly should be viewed as a last resort.

14.2 Poisson code in three dimensions

In the next example (downloadable, not included here in the notes), I have solved the model Poisson problem in three dimensions, with periodic boundary conditions in the x- and y-directions, and Neumann boundary conditions in the z-direction.

Exercise 14.1 *Compile and run the model three-dimensional code and benchmark its performance under OMP parallelism.*

Chapter 15

Handling data in large files

Overview

We examine new Matlab postprocessing tools to examine large three-dimensional files. We look at string maniuplation to postprocess files in which the data is not structured into neat, ordered columns.

15.1 Example

Consider the code `poisson_sor1.f90` from Chapter 14, and the resulting output files:

- `poisson.dat`
- `poisson_slice.dat`

The file `poisson_slice.dat` contains a slice of the three-dimensional array $C(x, y, z)$, at $y = L_y/2$, and can be viewed using the two-dimensional Matlab postprocessing tools we have constructed before. As well as viewing this file and plotting the result, it is also instructive to take a look at the data file itself. The first 10 lines of the file can be viewed in Linux by typing

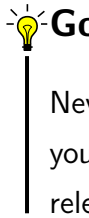
```
head -10 poisson_slice.dat
```

I get this:

```
>> bash-3.2$ head -10 poisson_slice.dat
  0.000000000000000E+000  0.000000000000000E+000  4.668321307964320E-003
  1.000000000000000E-002  0.000000000000000E+000  4.666004491564717E-003
```

```
  2.000000000000000E-002  0.000000000000000E+000  4.659109419590812E-003
  3.000000000000000E-002  0.000000000000000E+000  4.647589883399717E-003
  4.000000000000000E-002  0.000000000000000E+000  4.631510159553213E-003
  5.000000000000000E-002  0.000000000000000E+000  4.610833365189051E-003
  6.000000000000000E-002  0.000000000000000E+000  4.585632448160086E-003
  7.000000000000001E-002  0.000000000000000E+000  4.555879996932404E-003
  8.000000000000000E-002  0.000000000000000E+000  4.521657342747652E-003
  9.000000000000000E-002  0.000000000000000E+000  4.482946653571969E-003
```

Obviously, there are three columns containing the various values of X, Y, and C, which are then read into a Matlab file and reshaped into square arrays.



Good Programming Practice:

Never try opening a large file in its entirety - you will run out of memory and crash your computer. Instead, use tools like 'head', 'tail', 'more', and 'grep' to extract or view relevant information.

Next, we should try to write a Matlab postprocessing file to do the same thing for the full three-dimensional data file. However, before we try anything, we should not assume *a priori* that the data is ordered into columns (why should we?). Thus, I typed

```
head -10 poisson.dat
```

and got

```
>> bash-3.2$ head -10 poisson.dat
  0.000000000000000E+000  0.000000000000000E+000  0.000000000000000E+000
 -5.252804656865423E-002
  1.000000000000000E-002  0.000000000000000E+000  0.000000000000000E+000
 -5.250215979901304E-002
  2.000000000000000E-002  0.000000000000000E+000  0.000000000000000E+000
 -5.242439421169698E-002
  3.000000000000000E-002  0.000000000000000E+000  0.000000000000000E+000
 -5.229495753177844E-002
  4.000000000000000E-002  0.000000000000000E+000  0.000000000000000E+000
 -5.211384677229017E-002
```

Obviously, there are four columns containing the various values of X, Y, Z, and C. However, C has been pushed on to a second line! In short, in printing to files, unpredictable things can happen, and our Matlab file-reading tools need to be able to take account of these things.

I have written a Matlab file to take account of this line-jumping:

```
function [X,Y,Z,C]=open_single_dat_file_3d()

Nx=201;
Ny=201;
Nz=101;

n_lines=Nx*Ny*Nz;

filename='poisson.dat';
fid=fopen(filename);

X=0*(1:n_lines);
Y=0*(1:n_lines);
Z=0*(1:n_lines);
C=0*(1:n_lines);

for i=1:n_lines
    c1=fgetl(fid);
    vec_temp=sscanf(c1,'%f');
    x_temp=vec_temp(1);
    y_temp=vec_temp(2);
    z_temp=vec_temp(3);

    X(i)=x_temp;
    Y(i)=y_temp;
    Z(i)=z_temp;

    c1=fgetl(fid);
    vec_temp=sscanf(c1,'%f');
    c_temp=vec_temp(1);

    C(i)=c_temp;

    if(mod(i,100000)==0)
        frac=100*(i/n_lines);
        display(strcat(num2str(frac),'% done'))
    end

end

```

```
X=reshape(X,Nx,Ny,Nz);
Y=reshape(Y,Nx,Ny,Nz);
Z=reshape(Z,Nx,Ny,Nz);
C=reshape(C,Nx,Ny,Nz);

fclose(fid);

end
```

codes/poisson_code_threed/open_single_dat_file_3d.m

It is a simple extension to what has been done before, and exploits the fact that variable `fid` labels the current line in the open file, and that upon reading a line, the variable is incremented by one so as to label the next line.

Having read the data file into Matlab, the challenge is to view it in three dimensions. There is a useful **isosurface** feature in Matlab, which will plot a **level surface** of a three-dimensional function $C(x, y, z)$. Recall, that

$$C(x, y, z) = \text{Const.}$$

is a two-dimensional manifold surface embedded in $\mathbb{R}^3$, and can therefore be plotted. This is precisely the definition of a level surface. A analogue in $\mathbb{R}^2$ is a level line, or, in other words, a contour. There is a tremendous amount of machinery that comes with the isosurface function in Matlab, including how the surface is 'lit', and from what angle it is viewed. The best way to learn about this is to experiment, and use the 'help' pages. As a starting point for this experimentation, one can use the following code which genreates a level surface from the following **four** three-dimensional arrays: (X, Y, Z, C):

```
function []=make_iso(X,Y,Z,C,val)

fv=isosurface(X,Y,Z,C,val);

h=figure;
p=patch(fv);
set(p,'FaceColor','red','Edgecolor','none')
axis equal
view(30,30)
axis tight
camlight('headlight')
set(h,'Renderer','zbuffer');
lighting phong
grid on
set(gca,'fontsize',18,'fontname','times new roman')
xlabel('x')
```

```
17 ylabel('y')
18 zlabel('z')
19 drawnow
20
21 figfilename=strcat('isosurface_val','.fig');
22 saveas(h,figfilename)
23
24 end
```

codes/poisson_code_threed/make_iso.m

Here, 'val' is the level at which the isosurface is to be made. A sample result is shown in Figure 15.1

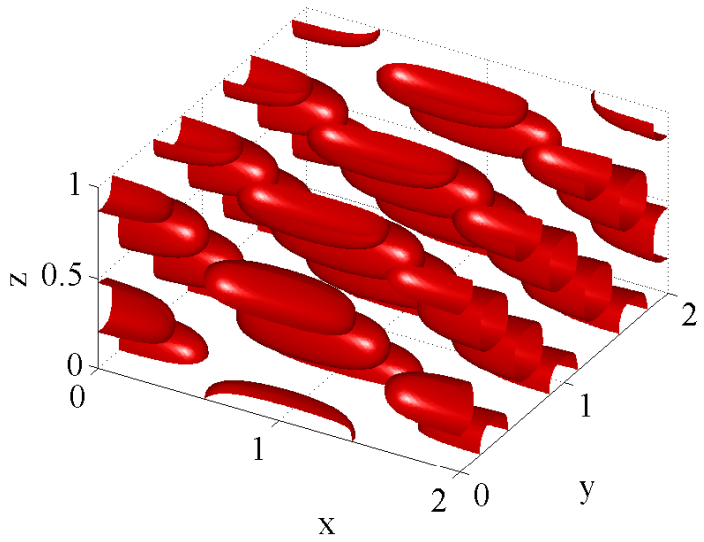


Figure 15.1: Isosurface for the Poisson problem, with $C = 0.015$.

15.2 Challenge problem

Exercise 15.1 *Write and execute a code to solve the three-dimensional diffusion equation whose steady-state solution is given by the Poisson problem discussed in this chapter. Postprocess the results in an appropriate fashion.*

Some notes:

- This will take a long time to run, and the results will involve a large amount of data.

- A family of isosurface plots might be the most appropriate way of showing the results. Some families of two-dimensional slices might also be appropriate.
- For the isosurfaces, the most appropriate isosurface level suitable for conveying the maxmum amount of information will vary over time. You should develop either an analytical formula or a numerical postprocessing tool to determine this level. Both approaches will involve studying the maximum of $C(x, y, z, t)$ over the domain, as a function of time.

15.3 Further challenge problem

Consider the following two files that can be downloaded from the website:

- File A: '3dchannel_ 7000.dat'
- File B: '3dchannel_ 65000.dat'

These files contain outputs from numerical simulations at different points in time. The simulations are from a two-phase Navier–Stokes solver. Ideally, the data should be structured into eight columns: $[X, Y, Z, U, V, W, P, \Phi]$, where X, Y, and Z are coordinates, U, V, and W are velocities, P is a pressure, and Φ is the so-called level-set function, which tracks which phase is which: with $\Phi < 0$ in the more viscous phase of the simulation, and $\Phi > 0$ in the less viscous phase. The numerical grid is of size $304 \times 152 \times 152$, and the files contain two header lines.

Exercise 15.2 *Use appropriate Linux tools to characterize the structure of the data in these files.*

In addition, the number of lines in each file can be estimated as follows:

```
wc -l filename
```

This should be equal to $304 \times 152 \times 152 +$ Number of header lines.

Now, as it turns out, in file A, the data have been output in a straightforward way, in strict column form.

Exercise 15.3 *Write a Matlab code to extract the data from file A, and to generate an isosurface plot, at* $\Phi = 0$.

Good Programming Practice:

For this assignment, it is a good idea to create two Matlab functions for the two sub-tasks that appear here. In this way, should the isosurface task fail, you will still have available data from the file-reading part of the task. This means that when you start over writing the isosurface task, you will be starting from a relatively advanced point. This will save a lot of time, given the intensity of the data-reading task to be performed.

Here is a snippet of a 'good' isosurface plot command for this problem:

```
fv = isosurface(X,Y,Z,Phi,0);
    p=patch(fv);
    set(p,'FaceColor','red','Edgecolor','none')
    axis equal
    view(30,30)
    axis tight
    zlim([0.1,0.5])
    camlight('headlight')
    set(h,'Renderer','zbuffer');
    lighting phong
```

Now, as it turns out, in file B, the data have been printed to the file in a truly bizarre way (which nevertheless saves some space; file B is smaller than file A):

1. For each gridpoint, the variables $[X, Y, Z, U, V, W, P, \Phi]$ extend over two lines in the output file.

2. Each piece of information is separated by a comma (not a space or a tab).

3. For each gridpoint, if a variable appears twice, it is stored only once. Thus, if at a certain gridpoint, $X = Y$, then instead of printing $[X, Y, Z, U, V, W, P, \Phi]$, $[2 * X, Z, U, V, W, P, \Phi]$ is printed instead.

4. Similarly, if $X = Y = Z$, then $[X, Y, Z, U, V, W, P, \Phi]$ is not printed; instead, $[3{*}x, U, V, W, P, \Phi]$ is printed.

Obviously, one could at runtime specify explicitly and strictly how the I/O is to be performed. However, I realised after the fact what had taken place, and it is wasteful to perform these large-scale simulations more than once, only to recreate the results a second time in a slightly-different

output format. Thus, it is necessary to write a Matlab script to account for all of these variations, and to gather the results into large three-dimensional arrays for isosurface plotting.

Exercise 15.4 *Write a Matlab code to extract the data from file B, and to generate an isosurface plot, at* $\Phi = 0$.

Hint: before doing so investigate the following built-in Matlab commands:

- `fgetl`
- `strrep` – for replacing characters in a string by new characters
- `strcat` – for joining two strings together
- `sscanf` – for reading strings into arrays of a specfied type

The result will be a beautiful picture like Figure 15.2.

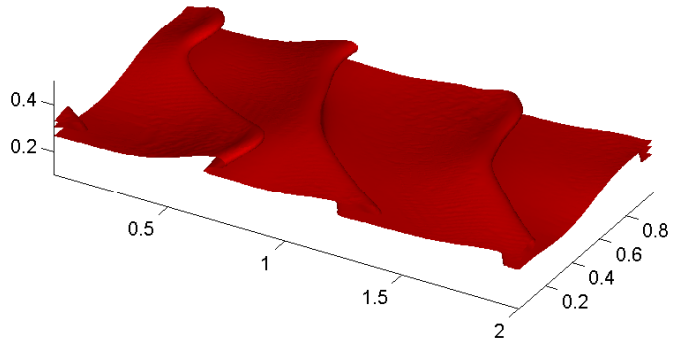


Figure 15.2: Isosurface plot at $\Phi = 0$, for file B

Appendix A

Facts about Linear Algebra you should know

Overview

In this appendix, let V be a real vector space with dimension $n < \infty$, and equipped with a scalar product

$$\begin{aligned}(\cdot|\cdot) : V \times V &\to \mathbb{R} \\ \boldsymbol{x}, \boldsymbol{y} &\mapsto (\boldsymbol{x}|\boldsymbol{y}).\end{aligned}$$

A.1 Orthogonality

- Two vectors $\boldsymbol{x}$, $\boldsymbol{y}$ are said to be orthogonal if $(\boldsymbol{x}|\boldsymbol{y}) = 0$.
- The set $\{\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n\}$ is called a **basis** for V if
  - $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n$ are linearly independent;
  - $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n$ span V.

  Thus, for any $\boldsymbol{x} \in V$, there exist real scalars $\alpha_1, \cdots, \alpha_n$ such that

$$\boldsymbol{x} = \sum_{i=1}^{n} \alpha_i \boldsymbol{x}_i.$$

- A basis $\{\boldsymbol{x}_i\}_{i=1}^n$ for V is said to be **orthonormal** if

$$(\boldsymbol{x}_i|\boldsymbol{x}_j) = \delta_{ij}.$$

Given an orthonormal basis $\{\boldsymbol{x}_i\}_{i=1}^n$, we have, for arbitrary $\boldsymbol{x} \in V$,

$$\begin{aligned} \boldsymbol{x} &= \sum_{i=1}^n \alpha_i \boldsymbol{x}_i, \\ (\boldsymbol{x}_j|\boldsymbol{x}) &= \sum_{i=1}^n \alpha_i (\boldsymbol{x}_j|\boldsymbol{x}_i), \\ (\boldsymbol{x}_j|\boldsymbol{x}) &= \sum_{i=1}^n \alpha_i \delta_{ij}, \end{aligned}$$

hence

$$\alpha_j = (\boldsymbol{x}_j|\boldsymbol{x}), \qquad \boldsymbol{x} = \sum_{j=1}^n (\boldsymbol{x}_j|\boldsymbol{x})\boldsymbol{x}_j. \tag{A.1}$$

In Quantum Mechanics, Equation (A.1) is called the **completeness relation**.

A.2 The Spectral Theorem

Throughout this section, let $V = \mathbb{R}^n$. The **usual basis** means

$$\begin{aligned} \boldsymbol{e}_1 &= (1, 0, 0, \cdots, 0, 0), \\ \boldsymbol{e}_2 &= (0, 1, 0, \cdots, 0, 0), \\ \vdots &= , \\ \boldsymbol{e}_n &= (0, 0, 0, \cdots, 0, 1). \end{aligned}$$

An arbitrary vector in $\mathbb{R}^n$ is written as a Cartesian n-tuple, $\boldsymbol{x} = (x_1, \cdots, x_n)^T$, which can be written in terms of the usual basis as

$$\boldsymbol{x} = \sum_{i=1}^n x_i \boldsymbol{e}_i.$$

In this section, we are interested in real, square $(n \times n)$ symmetric matrices; let $\mathbf{A}$ be such a matrix:

$$\begin{aligned} \mathbf{A} : \mathbb{R}^n &\to \mathbb{R}^n, \\ \boldsymbol{x} &\mapsto \mathbf{A}\boldsymbol{x}, \end{aligned}$$

such that $(\mathbf{A}\boldsymbol{x})_i = \sum_{j=1}^n A_{ij} x_j$. The symmetricness of $\mathbf{A}$ means that

$$A_{ij} = A_{ji}.$$

We have a simple lemma:

Lemma A.1 *The eigenvalues of* $\mathbf{A}$ *are real, and eigenvectors corresponding to distinct eigenvalues are orthogonal.*

A deeper result is the following:

Theorem A.1 *The eigenvectors of* $\mathbf{A}$ *span* $\mathbb{R}^n$ *and can be chosen to form an orthonormal set.*

This is a special case of the celebrated **spectral theorem** – the crowning achievement of Linear Algebra.

A.3 Diagonalizabillity

Here, we let $\mathbf{A} \in \mathbb{R}^{n \times n}$. For situations where the eigenvectors span $\mathbb{R}^n$, with

$$\mathbf{A}\boldsymbol{x}_i = \lambda_i \boldsymbol{x}_i, \qquad \mathcal{S}\left(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n\right) = \mathbb{R}^n,$$

then we can form the matrix

$$\mathbf{P} = \begin{pmatrix} | & & | \\ \boldsymbol{x}_1 & \cdots & \boldsymbol{x}_n \\ | & & | \end{pmatrix},$$

such that

$$\mathbf{AP} = \mathbf{PD}, \qquad \mathbf{D} = \begin{pmatrix} \lambda_1 & & \\ \vdots & & \vdots \\ & & \lambda_n \end{pmatrix},$$

Given the assumption that the eigenvectors span $\mathbb{R}^n$, the eigenvectors are all linearly independent, hence $\mathbf{P}^{-1}$ exists, and we have

$$\mathbf{P}^{-1}\mathbf{AP} = \mathbf{D}.$$

A.4 Jordan decomposition

Of course, not all matrices are diagonalizable. Sometimes (but not necessarily) the presence of a non-simple root in the characteristic polynomial may signal the non-diagonalizability of a matrix. One should note carefully that this is **not** a necessary condition for non-diagonalizability; e.g. $\mathbf{A} = \mathbb{I}$ is trivially diagonalizable, with a single repated root in the associated characteristic equation. Putting aside this discussion, in general, in situations of non-diagonalizability, there is a weaker result (i.e. weaker than the Spectral Theorem), namely the **Jordan decomposition**:

Theorem A.2 *Let $\mathbf{A}$ be any matrix in over the field of complex numbers. Then there exists an intertible matrix $\mathbf{P}$ such that*

$$\mathbf{P}^{-1}\mathbf{A}\mathbf{P} = \begin{pmatrix} \mathbf{J}_1 & & \\ \vdots & \vdots & \\ & & \mathbf{J}_k \end{pmatrix}, \qquad k \leq n,$$

where each $\mathbf{J}_r$ is a square matrix of the form

$$\mathbf{J}_r = \begin{pmatrix} \lambda_r & 0 & \cdots & \cdots & 0 \\ 1 & \lambda_r & \cdots & \cdots & 0 \\ \cdots & \vdots & & \lambda_r & \vdots \\ 0 & 0 & \cdots & 1 & \lambda_r \end{pmatrix}, \qquad r \leq n$$

with eigenvalues λ_r on the diagonal, ones just below the diagonal, and zeros everywhere else.

Note that the convention that the submatrices have 1s on the superdiagonal instead of the subdiagonal is also used, e.g.

`http://mathworld.wolfram.com/JordanCanonicalForm.html`

A.5 The operator norm

Definition A.1 (The L^2-norm of a matrix) *Let $\mathbf{A} \in \mathbb{R}^{n\times n}$ be a real matrix. We define the L^2-norm of $\mathbf{A}$ as follows:*

$$\|\mathbf{A}\|_2 = \sup_{\boldsymbol{x}\neq 0} \frac{\|\mathbf{A}\boldsymbol{x}\|_2}{\|\boldsymbol{x}\|_2}.$$

Throughout the rest of this course, we refer to the L^2 norm of a matrix as the **operator norm**.

Theorem A.3 (Properties of the operator norm) *Let $\mathbf{A} \in \mathbb{R}^{n\times n}$ be a real matrix. We have the following set of properties of the operator norm:*

1. *Positive-definite:* $\|\mathbf{A}\|_2 \geq 0$, $\|\mathbf{A}\|_2 = 0 \implies \mathbf{A} = 0$,
2. *Linearity under scalar multiplication:*

$$\|\mu\mathbf{A}\|_2 = |\mu|\|\mathbf{A}\|_2,$$

3. *The triangle inquality:*

$$\|\mathbf{A}+\mathbf{B}\|_2 \leq \|\mathbf{A}\|_2 + \|\mathbf{B}\|_2,$$

4. *Cauchy–Schwarz-type inequalities*

$$\|\mathbf{AB}\|_2 \leq \|\mathbf{A}\|_2\|\mathbf{B}\|_2, \qquad \|\mathbf{A}\boldsymbol{x}\|_2 \leq \|\mathbf{A}\|_2\|\boldsymbol{x}\|_2,$$

for all $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n\times n}$, $\boldsymbol{x} \in \mathbb{R}^n$, *and all* $\mu \in \mathbb{R}$.

Lemma A.2 *Let* $\mathbf{A} \in \mathbb{R}^{n\times n}$ *be a real matrix. Then,*

1. $\mathbf{A}^T\mathbf{A}$ *is symmetric;*

2. *The eigenvalues of* $\mathbf{A}^T\mathbf{A}$ *are non-negative.*

Proof:

1. We have
$$(\mathbf{A}^T\mathbf{A})^T = \mathbf{A}^T(\mathbf{A}^T)^T = \mathbf{A}^T\mathbf{A},$$
hence $\mathbf{A}^T\mathbf{A}$ is symmetric.

2. Let
$$(\mathbf{A}^T\mathbf{A})\boldsymbol{x} = \lambda\boldsymbol{x}.$$
We dot both sides by $\boldsymbol{x}$ using the ordinary dot product. For brevity, we use the following notation:
$$\boldsymbol{x}\cdot\boldsymbol{y} \equiv \boldsymbol{x}^T\boldsymbol{y} \equiv (\boldsymbol{x},\boldsymbol{y}).$$
We have,
$$\begin{aligned}\left(\boldsymbol{x}, \mathbf{A}^T\mathbf{A}\boldsymbol{x}\right) &= \lambda\left(\boldsymbol{x},\boldsymbol{x}\right),\\ (\mathbf{A}\boldsymbol{x}, \mathbf{A}\boldsymbol{x}) &= \lambda\|\boldsymbol{x}\|_2^2,\end{aligned}$$
hence, $\|\mathbf{A}\boldsymbol{x}\|_2^2 = \lambda\|\boldsymbol{x}\|_2^2$, and $\lambda \geq 0$.

Theorem A.4 (An explicit method to compute the operator norm) *Let* $\mathbf{A} \in \mathbb{R}^{n\times n}$ *be a real matrix. We have the following identity:*

$$\|\mathbf{A}\|_2 = \sqrt{\lambda_{\max}}$$

where $\lambda_{\max}$ *denotes the largest eigenvalue of the matrix* $\mathbf{A}^T\mathbf{A}$.

Proof: We have,

$$\begin{aligned} \|\mathbf{A}\boldsymbol{x}\|_2^2 &= (\mathbf{A}\boldsymbol{x}, \mathbf{A}\boldsymbol{x}), \\ &= \left(\mathbf{A}^T\mathbf{A}\boldsymbol{x}, \boldsymbol{x}\right), \\ &= \left(\boldsymbol{x}, \mathbf{A}^T\mathbf{A}\boldsymbol{x}\right). \end{aligned}$$

Now $\mathbf{A}^T\mathbf{A}$ is a real, symmetric $n \times n$ matrix so by the spectral theorem, the eigenvectors of $\mathbf{A}^T\mathbf{A}$ span $\mathbb{R}^n$ and are orthonormal. Thus, we can write

$$\boldsymbol{x} = \sum_i \alpha_i \boldsymbol{x}_i, \qquad \left(\mathbf{A}^T\mathbf{A}\right)\boldsymbol{x}_i = \lambda_i \boldsymbol{x}_i, \qquad (\boldsymbol{x}_i, \boldsymbol{x}_j) = \delta_{ij}.$$

Thus, we have

$$\begin{aligned} \left(\boldsymbol{x}, \mathbf{A}^T\mathbf{A}\boldsymbol{x}\right) &= \left(\sum_i \alpha_i \boldsymbol{x}_i\right) \cdot \left(\sum_j \lambda_j \alpha_j \boldsymbol{x}_j\right), \\ &= \sum_{ij} \alpha_i \alpha_j \lambda_j \delta_{ij}, \\ &= \sum_i \alpha_i^2 \lambda_i. \end{aligned}$$

Hence,

$$\|\mathbf{A}\boldsymbol{x}\|_2^2 = \left(\boldsymbol{x}, \mathbf{A}^T\mathbf{A}\boldsymbol{x}\right) = \sum_i \alpha_i^2 \lambda_i. \tag{A.2}$$

Let $\lambda_{\max} = \max_i \lambda_i$, and let $i_{\max}$ be the index of the maximal eigenvalue. The expression (A.2) is maximized by taking $\alpha_i = 0$ unless $i = i_{\max}$, i.e. $\boldsymbol{x} \propto \boldsymbol{x}_{i_{\max}}$. Note that this argument is not affected by the presence of eigenspaces of dimension higher than one, i.e. it is not affected by degenerate eigenvalues. Thus,

$$\|\mathbf{A}\boldsymbol{x}\|_2^2 = \alpha_{i_{\max}}^2 \lambda_{\max},$$

and

$$\frac{\|\mathbf{A}\boldsymbol{x}\|_2^2}{\|\boldsymbol{x}\|_2^2} = \lambda_{\max}.$$

In other words,

$$\sup_{\boldsymbol{x} \neq 0} \frac{\|\mathbf{A}\boldsymbol{x}\|_2}{\|\boldsymbol{x}\|_2} = \sqrt{\lambda_{\max}},$$

as required.